AR-009-738

DSTO-TR-0360

DSTO

A Review of Software Sizing for Cost Estimation

Gina Kingston and Martin Burke

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

A Review of Software Sizing for Cost Estimation

Gina Kingston and Martin Burke

Information Technology Division
Electronics and Surveillance Research Laboratory

DSTO-TR-0360

ABSTRACT

A variety of Software Sizing approaches have been conjectured in the literature. While many of these approaches have been used for Software Cost Estimation, none were specifically designed for Software Costing. This paper reviews current methods of Software Sizing to determine their suitability for Software Cost Estimation. The approach differs from previous approaches in that it is based on a framework of evaluation criteria including Measurement Theoretic as well as practical considerations.

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

Published by

DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury South Australia 5108

Telephone: (08) 259 5555
Fax: (08) 259 6567

AR No. 009-738
June 1996

APPROVED FOR PUBLIC RELEASE

A Review of Software Sizing for Cost Estimation

Executive Summary

This paper evaluates a variety of Software Sizing approaches found in the literature to determine their appropriateness for estimating the Cost of developing Software.

A framework of evaluation criteria is proposed which draws from Measurement Theory, the criteria proposed in both the Software Sizing and the Software Costing literature, and the authors' experience.

Individual measures of Software Size are evaluated against these criteria. The measures are broken into three classes: Length, Complexity and Functionality. The desirable properties are summarised for the measures in each of these classes.

None of the measures of Software Size investigated had all the desirable properties for a measure of Software Size for Cost estimation, and different desirable properties were held by the different classes of measure.

The authors conclude that different measures of Software Size should be used for different Cost Estimation activities. Estimation activities occurring early in the development of the system would normally use measures of the Functionality of the system, while later estimates could use measures of the Length of the source code.

Authors

Gina Kingston
Information Technology Division

Gina has been employed in the Software Engineering Group of the Information Technology Division of the Defence Science Technology Organisation (DSTO) since graduating from the University of Tasmania with a BSc with First Class Honours in Mathematics in 1990. She is currently undertaking a PhD in Software Costing through the University of New South Wales' School of Information Systems and working on the integrated Measurement, Assessment and Prediction of Software (iMAPS) task at DSTO.

Martin Burke
Information Technology Division

Martin has a BSc(Hons) in Physics, a MSc in Mathematical Statistics, and a PhD in Engineering Mathematics. He has held scientific and management positions at Rolls Royce (Aero), the SEMA Group Research Centre and the UK Atomic Energy Authority. Martin joined DSTO in 1991 as a section leader in Software Engineering Group and is the Task Manager of the integrated Measurement, Assessment and Prediction of Software (iMAPS) task. His current fields of specialisation include: Software Cost Prediction, Software Risk Assessment, and Safety Critical Systems and Software.

Contents

1. Introduction

> *Throughout the short history of the software industry, both researchers and practitioners alike have agonized over the industry's continuing failure to adequately control software development. That the problem is both serious, and very much alive is highlighted in a recent congressional report, in which eight major software projects in the U.S. Department of Defense experienced a combined cost overrun of more than one billion dollars.*
>
> [Abdel-Hamid, T. K. et al., 1993]

Accurate estimates of Software Cost are increasingly important. The Costs of developing and maintaining software systems are an increasingly large proportion of many companies' budgets [Abdel-Hamid, T. K. et al., 1993]. Therefore, organisations need to spend their software budget carefully, often choosing between systems according to their Cost to benefit ratio. Estimates which are too high may result in beneficial projects being shelved [Lederer, A. L. and Prasad, J., 1993a]. Conversely, estimates which are too low result in projects running over budget, or even being cancelled before completion.

Cost overruns of up to three times the original estimates are reported in [Heemstra, F. J., 1992], while a study of US organisations determined that 63% of projects over $50,000 overran and 14% of projects over $50,000 underran their estimates [Lederer, A. L. and Prasad, J., 1993b].

Despite these problems with cost overruns, the limited software budget of most organisations means there is still a need to choose carefully the systems to be developed. Even if these decisions could be made without a cost versus benefit analysis, these organisation could not afford the cost overruns which, given current estimation techniques, they are likely to incur. Thus, we need better estimation methods to both choose the systems to be developed, and to manage their development.

It is commonly accepted that labour Costs are the most difficult Cost to determine - some of the other Costs associated with developing software, such as the Cost of the hardware on which the software will be developed, can be accurately, and precisely determined a priori. Labour Costs constitute a significant proportion of the overall software development Cost. (It is stated in [SPC, 1994] that labour costs constitute 80% of the software development Costs.) Models for estimating labour Costs tend to determine the Effort required to develop the product, and then determine the Costs using this Effort and the average hourly, or monthly rates of the development staff.

Various methods for determining software development Effort were found in a study of 112 organisations, [Lederer, A. L. and Prasad, J., 1993a]. These methods were rated on a scale of 1-5 according to the extent of their use. A rating of 5 indicated that the method was used extensively. These methods included:

- guessing (2.76) and intuition (3.38),
- analogy, based on personal experience (3.77), or on documented facts (3.41),
- estimating packages (1.80), and
- statistical techniques, simple standards (2.33), or more complex statistics (1.49)

Current empirical and algorithmic methods of Software Cost estimation tend to relate development Effort to a measure of Software Size, and then compensate for the effects of secondary factors [SPC, 1994; Heemstra, F. J., 1992]. It has been show, [Kitchenham, B. A., 1992; Kingston, G. et al., 1995], that many of these other factors are not statistically significant. Thus Size measures are extremely important for estimating Effort, and therefore Cost.

Current Software Cost estimation methods are often seen as poor predictors of Effort [Kusters, R. J. et al., 1990; Kok, P. A. M. et al., 1990]. This is not surprising as most of the current measures of Size were not specifically designed for Effort estimation. It is proposed by the authors, that a measure of Software Size which is specifically designed for Effort estimation would have a better correlation with Effort than existing models. Such a measure of Size could form a sound basis for developing more complex models of Software Cost, and could help ensure that all other factors used in the models were independent of the measure of Size.

With this in mind, current Software Sizing methods are reviewed in the context of their suitability for Effort Estimation. A framework of criteria derived from Software Measurement Theoretic considerations is used as the basis of the review. (An introduction to Software Measurement Theory can be found in [Fenton, N., 1991].) The insight afforded by this approach gives the opportunity for consistent appraisal of the significance to Software Costing of alternative Size Measures. The authors believe that, as the first published review of its kind, it represents a valuable contribution to Software Engineering. iMAPS planned future work in Software Costing will allow this to be exploited for the ADO's benefit.

This paper draws on previous reviews of Software Size which have not had the same focus on Effort estimation and measurement theory, in particular [Lokan, C., 1995; Lokan, C. J., 1993; SPC, 1994; Hastings, T., 1995], as well as work on criteria for the comparison of Effort Estimation models [Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, 1990; Heemstra, F. J., 1992; Kitchenham, B. A. and Taylor, N. R., 1984; Kusters, R. J. et al., 1990; Hufton, D. R., 1985].

Outline

This paper is a review of the appropriateness of different measures of Software Sizing for Effort Estimation.

For readers with limited time, a direct comparison between the measures considered is given in Section 10. This section discusses the significance of the differences between the measures. The suitability of the evaluation criteria and future research directions are also discussed.

The criteria for the comparison and evaluation of different Sizing approaches are given in Section 3. These include all published criteria of which the authors are aware as well as additional criteria proposed by the authors. The criteria include the purpose and the scope of the approaches, which are discussed further in Section 2. This discussion considers the nature of software, and what is, or can be meant by, Software Size.

Sections 4 to 8 evaluate individual measures of Software Size according to the criteria proposed in Section 3. The two best known measures, Lines Of Code (Section 4) and Function Points (Section 6) are discussed in their own sections. The other sections discuss, Other Code Based Measures (Section 5), Other External Measures (Section 7) and Measures which combine Code and External sub-measures (Section 8). Measures of Software Growth are discussed in Section 9.

2. Measures of Software Sizes

> *Measurement activities must have clear objectives or goals, and it is these which will determine the kinds of entities and attributes which must be measured.*
>
> [Fenton, N., 1991]

Software Size measures have been used for a range of activities (and therefore to achieve different goals and objectives). These include:

- comparing the productivity of different development tools, development technologies and developers [Tate, G. and Verner, J. M., 1991],
- comparing similar software systems,
- understanding software systems,
- estimating development Effort,
- estimating maintenance Effort,
- estimating the time to complete a system whose development has commenced,
- determining which, if any, systems should be developed [Tate, G. and Verner, J. M., 1991],
- estimating reliability [Friedman, M. A. et al., 1995],

- tracking the Size of a project [Tate, G. and Verner, J. M., 1991],
- tracking changes in the estimated Effort, Cost and Schedule of the project,
- determining the impact of these changes [Tate, G. and Verner, J. M., 1991], and
- scheduling activities and project planning [Tate, G. and Verner, J. M., 1991].

As the nature of these activities varies, it is understandable that they may not all be correlated with the same measure. Therefore, there exists a variety of measures of Software Size.

However, measures of Software Size developed for one purpose have often been used for other purposes without due consideration of the implications. Common problems which arise from this include: missing important information; including spurious information; and inappropriate scales (see also Section 3.2). Consequently, the meaning of the values of the measure is often misinterpreted with varying impact depending on context.

Furthermore, it is not clear whether a measure of Software Size should be used for these activities. Some specific questions which arise from this are:

- Which of these activities require a measure of Size?
- What type of Size measure is required?
- Should the Size of the Software, the Software System, or the Development be measured?

To answer these questions, there is a need to examine what is meant by the terms software, software system and development.

The IEEE definition [IEEE Std 610.12, 1990] is:

> software — Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

While this definition covers many of the aspects of software, it is not clear whether it only pertains to artefacts (tangible products of software development, such as source code and test plans), or also to the relationships between the artefacts, to the use of these artefacts and/or to the development process. Furthermore, it is not clear which artefacts from the development process are included. For example, should test procedures or user's manuals be included as part of the software. The following definitions are preliminary may be refined on the basis of later analyses.

Definition 1: Source Code

> Source Code is a collection of instructions for a computer in the language in which they were first written or generated. That is, the instructions are not the result of a compilation into a machine-readable form.

The Source Code of a Software System includes any Source Code developed or used during the development of the system (See Definition 2 Definition 4). It specifically includes: the Source Code to be delivered to the clients or users (the Delivered Source Code), the source code to any automatic code or test generators developed for use on the project and any test procedures developed.

The Source Code which delivers the required functionality is called the Primary Source Code.

The Source Code of any automatic code generators and any test procedures is called the Secondary Source Code.

The Source Code of automatic test generators, and prototypes is called the Tertiary Source Code.

Definition 2: Software Development

Software Development is the process of producing Primary Source Code to fulfil a set of explicit or implicit requirements.

Definition 3: Software

Software is defined as any artefact of a Software Development. It specifically includes the Source Code, Object Code and Documentation.

The Primary Software includes the Primary Source Code and related Primary Programs, the User's and Training Manuals, and the initial data in the system.

The Secondary Software includes the Primary Software and the Secondary Source Code and Maintenance Manuals.

The Tertiary Software is all the Software produced as part of a Software Development. This specifically includes the Primary and Secondary Software.

Definition 4: Software System

A Software System has both physical and intangible components. It is all the artefacts (Software) produced during a Software Development and the relationships of those artefacts with each other, the Development process, the requirements, and the hardware and humans with which the Primary Programs must interact.

The Primary Software System of a Software System is the Primary Artefacts (Software) and their relationships.

The Secondary Software System of a Software System is the Primary and Secondary Artefacts (Software) and their relationships.

The Tertiary Software System of a Software System is identical to the Software System.

Different activities will require the Size of different components to be measured. Intuitively, planning and tracking activities would require information about the Size of the artefacts in the Tertiary Software System and Users would be more interested in the Size of the Primary Software. Effort Estimation could be best done with knowledge of the Tertiary Software, or possibly of the Tertiary Software System, but with the Size of the Primary and Secondary Software being important components.

Having considered what we mean by Software, before we define its Size, we need to consider what, in general, we mean by Size.

Definition 5: Size [Sykes (Ed.), J. B., 1977]

> Relative bigness, dimensions, magnitude.

We are now ready to think about what should be meant by the Size of a Software (System) for Effort estimation.

Intuitively, the Effort to develop a Software System can be thought to consist of three possible components: a fixed startup amount; an amount to actually write, or type, the software; and amount to understand and design the System. These last two components give rise to three different classes of Software Size measures.

The first measures are of class Length. These typically measure the Size of the System in terms of the length of the (Primary) Source Code of the program. Effort estimates based on Length measures are common. In these models, the Effort to understand the program and write the other Software artefacts is assumed to be proportional to the Length of the Source Code.

Complexity measures try to capture how difficult it is to understand a Software System. Typically, they look at the Complexity of the components of the Source Code. The Complexity of the whole system is not considered, and they are not generally used for Effort Estimation without another measure of Software Size.

A third class of Software Size measures which are commonly used are Functionality measures. These measures try to capture what a system should do. The Effort to develop the system, and sometimes the Length of the System are assumed to be proportional to the amount of Functionality in the System. (Alternatively, it may be assumed that the Effort due to the Functionality dominates the total Effort.) The Effort Estimation models often include additional factors, such as complexity factors to try to improve the predictive power of the models.

Definition 6: Software Size

> A Software Size measure is any measure of the relative bigness of the Software System which reflects one or both the length of the Software System and the difficulty of understanding and designing the Software System.

One problem with this definition is that many of the Software Size measures it defines depend on how the code is written. For example, the Length of the Source Code and the Complexity of the Source Code depend on how the system is

implemented. Also, the Effort to understand and design the system can depend on the novelty of the application, and on the development process (eg the quality of the development team and the techniques used). Ideally we want a measure of Software Size which is independent of how the software is developed. We call this Software Scope.

Definition 7: Software Scope

> A Software Scope measure is a measure of the relative bigness of a Software System under standard assumptions. These assumptions should ensure that systems developed to the same requirements at different times and locations have the same Software Scope. Where necessary, a standard development process, development environment and application novelty should be assumed. Under the assumed conditions, the measure should reflect one or both of the length of the Software System and the difficulty of understanding and designing the Software System.

This paper considers properties of Software which have been considered or used as measures of Software Size. Particular attention is paid to measurement theory, and the use of Software Size for Effort estimation. Most of the Software Sizes considered are not measures of Software Scope. This is discussed further in Section Future Directions.

3. Criteria for Comparing and Evaluating Software Sizing Techniques

For consistency, a systematic approach is required for the comparison and evaluation of different approaches of Software Sizing.

A framework for the comparison and evaluation of Software Costing methods was proposed. This uses nearly all the criteria (of which the authors are aware) from earlier reviews of Software Sizing (eg [Lokan, C. J., 1993]) and criteria commonly used for the evaluation of Software Costing models. These last criteria were included because this evaluation is concerned with the use of Software Sizing for Software Costing.

The framework was reviewed for consistency and completeness, and additional criteria were proposed to strengthen the framework. The final framework used in this paper is given in Figure 1. However, future work may further refine the framework. The appropriateness of the criteria is discussed further in Section 10.

The current framework consists of three high-level criteria: the basis, both practical and theoretical from which the approach was formed; measurement theoretic aspects; and the usefulness of the measure in practice. These criteria where chosen to accord with the measurement theoretic approach of this review. Measurement (theory) aspects were thus an obvious consideration. The Basis criterion provides information on the context in which the measurement was developed, such as its purpose. The final high-level criterion, Use, looks at the practical aspects of

implementing and using the measure. As these criteria cover the background, theory and practical aspects of Software Sizing, the authors believe that no additional high-level criteria should be required.

The high-level criteria are broken down in the following sections.

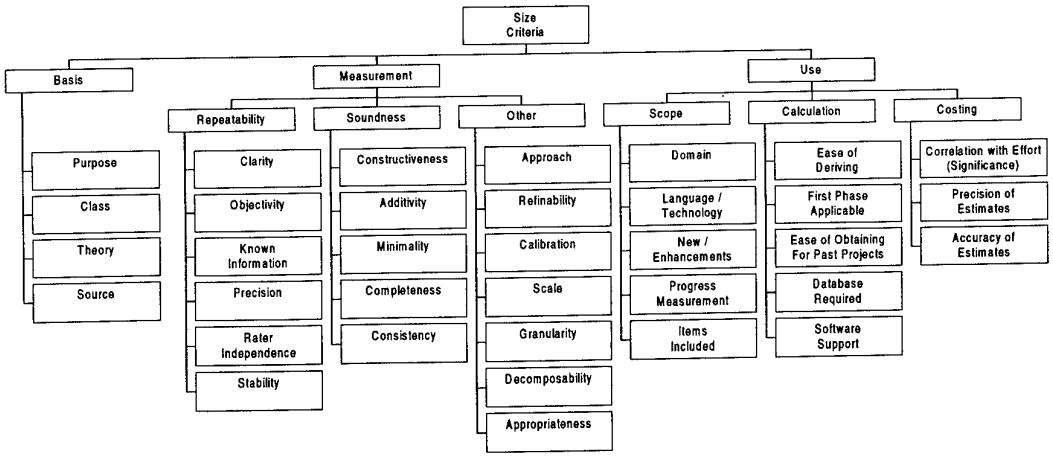

Figure 1: A Framework for the Comparison and Evaluation of Software Sizing Approaches

3.1 Basis

This criterion covers why (Purpose) and how (Theory and Source) the Sizing techniques were developed. A final sub-criterion, Class, is a classification mechanism is included under Basis as it is related to the Purpose and does not fall into any of the other high-level criteria.

Purpose

Software Sizes have been proposed for a variety of purposes (Section 2). The usefulness of a particular Size measure depends on how well the Size measure fits the purpose for which it is being used. It may not be *meaningful* [Fenton, N., 1991] for a measure of Size developed for one purpose to be used for another.

As no known Size measures have been proposed explicitly for Effort estimation we need to consider why they were proposed, in order to determine if they are meaningful for Effort explanation and estimation. Most Size measures will have strengths and weaknesses for Effort Estimation.

Class

Measures developed for different purposes need not be different. For example, to determine if a car will fit in a parking space (its length), or under a bridge (its height), both require a measure of distance. In this document, the property to be measured, as opposed to the purpose of the measurement, is called the *Class*.

For Example:

> Our purpose may be to determine whether a person who has been consuming alcohol is capable of driving safely (their Intoxication).
>
> Two tests which have been used to measure Intoxication are: 1) to check the amount of alcohol in the person's blood and 2) to see if they could walk in a straight line. The related measures are of Class Blood Alcohol Concentration and Straight Line respectively.
>
> Measures of Class Straight Line might also be used (purpose) to assess aspects of the quality of the writing of young children learning how to print.

Class is a refinement of previous classification schemes. Different classes of Software Size are considered in both [Lokan, C. J., 1993] and [Fenton, N., 1991]. However, neither define a criterion analogous to Class. Fenton's term, *Attribute*, (which is also used by Lokan) is defined as "the feature or property of the entity which we are interested in". It is similar to Class, but would make a distinction between Length and Height.

It should be noted that the term Class is used in a manner similar, but different, to that used in the context of Object-Oriented systems development. Alternative terms, such as Type, were considered. However, the name Type was considered inappropriate as the term is more heavily overloaded in Computer Science than Class, and its meaning is significantly different to that used in this context. Conversely, Measurement Classes could sensibly be modelled in a system by using Object-Oriented Classes.

Three Classes of Software Size measures are considered. Two of the Classes, namely Length and Functionality, were used in [Lokan, C. J., 1993]. We also consider a third class: Complexity [Fenton, N., 1991]. While Complexity is often not thought of as a measure of Software Size [Lokan, C., 1995], it is possible that it is a more appropriate measure, for the estimation of Software Cost, than Length or Functionality.

It should be noted that most of the measures considered in this paper consider the concept of Software to be related to the Source Code of the system. They do not really capture either the data or the documentation aspects (Section 2). While this is a valid view, with the data and documentation being considered as functions of the Size of the Source Code, additional classes of measures may be required to capture all aspects of Software. That is, the class may depend on the artefact(s) being measured.

Theory

An underlying theory assists in reasoning about, refining, and extending a measure.

Just as there are several scales for temperature, there can be several scales for the different classes of Software Size measures. An underlying theory is required:

- to compare measures made on different scales

- to compare measures made on the same scale,
- and to reason using these measures.

Software Sizes are often used with little or no regard for the underlying theory which determines how they may be legitimately used. This is true when measures are used as inputs into Software Costing models, where the measures are assumed to have the same properties as the standard "length" of an object. In particular, it is assumed to be a ratio scale measure. It is also true that a theory is required for Software Sizing models, particularly where the Size of the Software is assumed to be a function of more primitive measures.

For Example:

> As mentioned previously, a person's level of Intoxication is often inferred from their Blood Alcohol Concentration.
>
> It is now common for Random Breath Tests to be used to obtain an indication of a person's blood alcohol. If the reading is sufficiently high, then the more inconvenient blood tests can be administered.
>
> Random Breath Tests were developed using a theory which relates Blood Alcohol Concentration to Breath Alcohol Concentration. Similarly, a theory is required to link Blood Alcohol Concentration to Intoxication.
>
> (It has been shown that the amount of alcohol in a person's blood increases the likelihood of an accident. The risk of having an accident when you have a Blood Alcohol Concentration of 0.05 is double that of when you have no alcohol in your system [Camelotti, W., 1995]. However, it is also known that alcohol affects people differently. So while this is generally true, this risk of an accident may not double for a particular individual. Thus there is still some opposition to the use of Blood Alcohol Level as a measure of Intoxication.)

Many researchers have tried to justify their models using statistics, or intuition. However, statistics is only applicable if the data being analysed is appropriate and intuition is only a fore-runner to theory. Lack of an underlying theory may result in weaknesses in the model. For example, COCOMO has several theoretical weaknesses, [Kitchenham, B. A., 1992].

Source

Two different approaches to Software Sizing have been developed [Lokan, C. J., 1993; Levitin, A. V., 1987]: measures of Software Size which are based on information available prior to coding (external measures), and methods for estimating Software Sizes which are based on the source code (internal measures).

Until the 1980's most measures of Software Size were internal. Thus, many Software Cost Explanation models - and many models for determining programmer productivity - are based on internal measures of Software Size.

To use a Software Sizing method for Software Costing (Effort Estimation), we need a measure of the Software Size which can be determined early in the Software Development. This, in part, motivated the development of external measures of Software Size, which are generally available earlier than internal measures. Many of the newer Software Costing approaches are based on these external measures of Software Size.

It is possible that future measures of Software Size will be derived from alternative sources. For example, measures which are based not only on the Source Code, but also on other artefacts - such as documents - would not fall into either category based on the definitions above.

3.2 Measurement

The authors believe that it is preferable for Software Sizing and Software Costing approaches to be consistent with measurement theory [Fenton, N., 1991]. If approaches are developed to satisfy the axioms of the theory then their strengths and the limitations can be reasoned about in a repeatable, rigorous manner, and meaningfully applied to the problem at hand.

For Example:

> Consider two people, Xavier and Yvonne, who after sharing a bottle of wine at a conference dinner have their blood tested for alcohol. Suppose that their Blood Alcohol Concentrations are measured at 0.05 mL/L and 0.04 mL/L respectively.
>
> We can reason within the legal Drink-Drive model that Xavier is unfit to drive home, whereas Yvonne is entitled to drive. However, we need a model with a broader scope and which uses further measurements to test our intuitive conclusion that Xavier drank more than his fair share of the wine, or to conclude that he is less capable of driving safely than his drinking partner!
>
> In general, careful consideration of Measurement Theory is necessary when a measure is used for a purpose other than that for which it was developed.

There are many aspects to measurement theory. The breakdown proposed in this paper is to consider if the measure is repeatable; the soundness of the measure, from a measure theoretic point of view; and other measurement theoretic considerations, such as the scale of the measure and the approach(es) used to obtain values for the measure. The first three criteria are included as they are considered important aspects of measurement theory. The remaining criterion is provided for completeness. In future versions of the framework (Figure 1), additional criteria for Measurement Theory criteria may emerge from the Other criterion if their perceived importance increase.

3.2.1 Repeatability

This criterion [Kitchenham, B. A., 1992; Lokan, C., 1995] addresses the need for a measure to be used consistently, by the same, and by different people. The sub-criteria address these concerns both directly, and by considering the factors which influence them. The criteria consider the effect of: what is to be measured (Clarity); how it is to be measured (Known Information and Objectivity); by how much the value of the measure may vary (Precision and Stability); and by whom the measurement is made (Rater Independence). The criteria do not directly consider how the purpose of the measurement may change the values obtained, as the measurement are assumed to be of Software Size for Effort Estimation.

Clarity

A primary cause for different values being obtained for the same system (by different people) is that the definition does not clearly define what is to be measured. That is, that the definition is ambiguous [Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, F. J., 1992; Hufton, D. R., 1985].

For example: Lines Of Code has an obvious, intuitive, meaning. However, it is hard - if not impossible - to define a non-ambiguous measure for Lines Of (source) Code which is applicable to all languages.

This criterion covers the Clarity of the definition of what is to be measured. Other Repeatability criteria cover aspects of how, when and by whom the measure should be used.

Objectivity

A related cause of variations in measurement values is that some measures are subjective rather than objective [Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, F. J., 1992]. For example, determining the level of Intoxication of a driver by looking at how straight a line they can walk is a subjective measure.

Subjective measures may be given different values by different raters, and may even be given different values by the same rater in different circumstances.

Even measures which can be measured objectively are often measured subjectively. Consider the problem of determining which of the two lines in Figure 2 is longer. On first glance, the second line appears longer. However, when measured objectively it can be seen that the two lines are the same length.

Figure 2: An Optical Illusion

Known Information

Sometimes subjective estimates of essentially objective measures arise because of lack of information about their true values. For example, estimates of Lines Of Code are often required by Software Costing methods prior to coding. Therefore, Sizing and Costing methods which rely only on known information are preferred [Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, 1990; Heemstra, F. J., 1992; Kusters, R. J. et al., 1990].

Boehm and Wolverton called this criterion Prospectiveness. The authors call it Known Information.

Precision

Most measures of Software Size imply that the value obtained is the true value and do not qualify the precision or the uncertainty of the value. Where the measures must be estimated, determined using subjective information, or contain information which is likely to change with time, their precision should be quantified.

For example, a value for the number of Lines Of Code obtained prior to coding is less precise than the value obtained after coding, which, provided that the measure is properly defined, can be totally precise.

If we know the precision of the estimate, we can state how certain we are that the true value lies in a given range.

It is also useful to determine how estimates are changing with time. While estimates tend to become more precise with time, we cannot always assume this. We should provide an indication of the likelihood of any estimate, or range of estimates, being correct. This enables us to determine if the project is becoming better defined, and less risky, or if the risk associated with the project is increasing.

Rater Independence

Measures which are objective, unambiguous, and use known information should result in different raters (the people taking the measurements) arriving at the same value, or range of values, for a metric.

Even where these three conditions do not hold, the difference between raters may be negligible. This is called Rater Independence and can be tested experimentally.

Measures which are (almost) Rater Independent [Kemerer, C. F., 1993] are preferred as they provide a better basis for the prediction of other attributes.

Stability

A criterion related to Rater Independence is Stability, or Sensitivity [Kitchenham, B. A. and Taylor, N. R., 1984; Heemstra, 1990; Heemstra, F. J., 1992]. A measure is Stable if small variations in its sub-measures result in small variations to its value. This is particularly important if the sub-measures are subjective, estimated, or otherwise imprecise.

3.2.2 Soundness

As well as being repeatable, Software Size measures should be sound. That is the measures should be complete and, internally and externally consistent. Two types of criteria for soundness are used in this paper. The first type includes Constructiveness and Additivity. These criteria were specifically proposed for Software Sizing or Effort Estimation. They evaluate the soundness of the measure against experts' intuitions on Software Sizing. The second set of criteria are those the authors believe are generally applicable to measures. They are Minimality, Completeness and Consistency.

Constructiveness

This property has been given many names:

- constructiveness [Boehm, B. W. and Wolverton, R. W., 1980],
- traceable [Heemstra, 1990; Heemstra, F. J., 1992], and
- logical [Hufton, D. R., 1985].

A measure is constructive if variations in its value can be simply explained by (or traced to) variations in its sub-measures.

Additivity

Additivity is concerned with how the measure of an item, in this case a piece of software, relates to the measures on its components. More precisely, it is concerned with whether (and how) the measures on the components can be combined to determine the measure on the original item.

For example, how should the Size of a system, composed of two sub-systems (A and B) which may have a common component, be calculated. (See Figure 3).

It may be that the Size of the system:

- cannot be determined from the sub-systems A and B,
- depends on the Size of the common component
- is the sum of the Sizes of the two components,
- is at least the sum of the two components,
- is at most the sum of the two components.

Any of the last three may be appropriate depending on the purpose of the Size measure.

A measure is Additive (or summable [Symons, C. R., 1988]) if the Size of a system is at least the sum of its sub-systems when there are no common components. The desirability or otherwise of Additive measures is discussed in Section 10.1

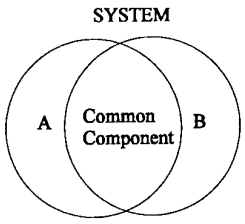

Figure 3: Venn Diagram of a Composite System with a Common Component

Minimality

All the sub-measures of a measure should be necessary for the measure to have the appropriate properties (parsimony, [Boehm, B. W. and Wolverton, R. W., 1980]). That is, if any of the sub-measures were removed, then the meaning of the measure would change. For example, if, when a sub-measure was removed, the relative Sizes of two systems changed then the measure is minimal. A set of measures which are not minimal are said to contain a redundant measure.

In addition, the sub-measures should be independent from each other, and should not be highly correlated with each other. If the sub-measures are highly correlated, then they should be replaced by an alternative set of measures which cover the same properties, but are not highly correlated. When the sub-measures are highly correlated, it is difficult to determine the effect of each sub-measure on the measure. This can effect Constructiveness, Stability, and the ability to check consistency and completeness.

This criterion is used to determine if the measure is Minimal. That is, there is no redundancy in the measure so each component in the measure is necessary.

Completeness

A related criterion to Minimality, is Completeness [Heemstra, 1990; Heemstra, F. J., 1992; Kusters, R. J. et al., 1990]. That is, does the measure capture everything it was intended to capture. While it is often possible to identify incomplete measures, it is difficult to ascertain whether a given measure of a property like Software Size is Complete. In this case, we are interested in if it is complete enough to be used for Effort estimation. The criteria for this are covered in Sections 3.3.1 and 3.3.3.

Consistency

This criterion considers whether the meaning of the values of the measure are consistent. That is, can the values obtained for different systems be compared?

For example, does the Language or Application Domain need to be the same for valid comparisons between the values for Lines Of Code for two systems?

3.2.3 Other Measurement Properties

Measurement properties which are considered in this section are the criteria: Approach, Calibration, Refinable, Scale and Granularity. These criteria are concerned with how the values of the measure may be obtained (Approach and Calibration), refined (Refinable) and manipulated (Scale) or the level of distinction between systems which is possible (Granularity). They were all determined from the literature.

In the future, additional criteria may be added to Other Measurement Properties, and some of the criteria in Other Measurement Properties may be promoted to sub-criteria of Measurement Theory.

Approach

The 'measurement', or estimation approaches used for Software Sizing are not equally valid according to Measurement Theory. One reason for this is that many of the so-called measures of Software Size were not developed with due consideration for Measurement Theory.

Five approaches to measurement or estimation which are applicable to, or have been used for, Software Sizing have been identified from estimation literature. These are expert opinion, analogy, algorithmic and empirical [Lokan, C. J., 1993; Kitchenham, B. A. and Taylor, N. R., 1984]. The fifth approach which can also be used to determine Software Sizes is direct measurement [Fenton, N., 1991]. The definitions used for these terms in this paper are given below. However, it should be noted that the meaning of the terms empirical and algorithmic are slightly different to those which may be used in other papers.

Expert Opinion: Expert Opinion is the process of obtaining an estimate of Software Size based on the instincts of the (suitably experienced) person making the estimate. There is no direct comparison with other Software, nor is there an explicit model for obtaining the estimate. This approach should only be used early in the development of a measure. Once the measure is understood, a more rigorous approach should be used to calculate its value.

Analogy: Analogy is the process of obtaining an estimate of Software Size by comparing the Software system of interest to other systems of known Size. The type, and proportion of differences and similarities are used to determine the Size of the new system, from the Size of the previous systems. This approach should only be used early in the development of a measure. Once the measure is understood, a more rigorous approach should be used to calculate its value.

Algorithmic: An algorithmic process is one by which an estimate of Software Size is obtained by determining the values of sub-measures and using an equation to determine the Size of the new Software system. In this criterion, the term will only be used where the

equations are derived from expert opinion or analogy. That is approaches which are not correct with respect to Software Measurement Theory. However, in general, it also applies to approaches which are valid measures. In this paper, such approaches are described as empirical.

Algorithmic approaches are not considered very highly: if the components of the measure are well-enough understood to be fixed, then it should be possible to determine their weights empirically.

Empirical: An empirical process is one by which an estimate of Software Size is obtained by determining the values of sub-measures and using an equation calibrated using historical data, to determine the Size of the new Software system. In this paper, the term is only used for valid measures, while in general it is used even when the approach is not correct with respect to Software Measure Theory. In this paper, such approaches are described as algorithmic. Measures whose values are derived using an empirical process are also called Indirect Measures.

Direct: Direct measurement is one which can be determined without reference to intermediate measures. An example of a direct measurement is the assessment of Lines of Code from the code of a Software System.

Calibration

When measurements are determined using empirical approaches it is often necessary to calibrate the approach for different development environments [Heemstra, 1990; Heemstra, F. J., 1992; Kusters, R. J. et al., 1990]. This arises from the need to obtain sufficiently accurate or precise estimates for their purpose. This criterion considers whether calibration is necessary and/or possible for measures of Software Size (as opposed the calibration of Effort Estimation model based on the Software Size measures). If it is necessary, then the criterion considers the requirements for calibration.

Refinable

When using Software Sizing for Software Costing, it is desirable to obtain estimates of the Software Size (and Cost) early in the development, and to refine them as the development progresses. Estimates are required early for use in planning and decision making. However, such estimates generally contain a high level of imprecision as the full details of the project are often not available early in its development. Therefore, estimates of Size (and Cost) should be refined during the development. This criterion looks at the ease with this can be done.

Scale

The scale of a measure determines the type of operations which can be performed on it [Fenton, N., 1991].

Nominal: The measure is used to classify the objects it characterises. Each item is associated with a particular name. Only equality and inequality operations can be performed on the measure values.

Ordinal: The values of the measure are ordered. Comparison operations can be performed on the measure values.

Interval: The values of the measure are ordered, and it is meaningful to perform operations such as addition and subtraction on the measure.

Ratio: There is a fixed zero and the measure is also an interval scale. Operations such as multiplication and division are allowed.

Absolute: All values of the measure are fixed. An absolute scale is also a ratio scale.

The scale of a measure also affects the types of statistics which can be performed on the measure. The scale should be ratio [Lokan, C., 1995] or absolute if standard techniques, such as linear regression are to be applied.

This criterion identifies the scale of the measure.

Granularity

The Granularity of a measure is the coarseness of its possible values. In the early stages of a Software development, it is likely that only a coarse (low resolution) measure of the Software Size is possible, while later, finer (high resolution) measures of Size may be possible. (See also Refinable.) This criterion is often incorrectly called scale.

For Example:

Consider the two measures of Intoxication introduced on Page 9: Blood Alcohol Concentration and Straight Line (which indicates whether or not the person can walk in a straight line).

The measure Straight Line has two possible values: Yes and No. Its granularity is coarse.

The measure Blood Alcohol Concentration has a range of values starting at 0.00 and increasing in steps of 0.01 to a practical limit of around 0.25. (The step size is limited by the accuracy and precision of the tools available to perform legal Blood Alcohol tests.) Thus Blood Alcohol Concentration has a much finer granularity.

As Blood Alcohol Concentration has a finer granularity, it provides you with more information. For example, one advantage of the Blood Alcohol Concentration measure is that you can use it to estimate how much you can drink and remain legally able to drive - provided you can remember how much you'd had to drink when the reading was taken.

The Granularity criterion has also been called:

- Detail [Boehm, B. W. and Wolverton, R. W., 1980] and
- Quantification [Heemstra, F. J., 1992])

Decomposable

This criterion considers the performance of the measure when used to estimate the Size of components of the system [Hastings, T., 1995]. One reason for considering whether the measure is Decomposable is that it is important in project tracking. (See also Refinable.)

Appropriateness

This criterion is used to consider if the Size measure is appropriate for Software Costing.

3.3 Use

This criterion covers areas which affect how Software Size measures may be used, particularly for Software Costing. It indicates when and where the measure is applicable (Scope), how easy it is to calculate (Calculation) and how useful it is for Software Costing. What is measured; why it is measured; how it is measured; and by whom it can be measured; are not considered under Use. Some aspects of these issues are discussed under Clarity (Section 3.2.1); Purpose (Section 3.1); Known Information (Section 3.2.1); and Rater Independence (Section 3.2.1), respectively. Other aspects are discussed in the general description of the measure or are not discussed due to lack of information in the literature.

Hastings, [Hastings, T., 1995], also proposed Acceptance and Validation criteria. That is, is the measure widely used in industry, and has the measure been validated. These have not been explicitly considered in this paper for two reasons. Firstly, because they do not provide a true indication of the worth of the measure from a measurement theoretic point of view, and secondly because the information is not readily available in the literature.

3.3.1 Scope

The scope of a Software Size measure determines those systems which can be Sized using the measure [Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, 1990; Heemstra, F. J., 1992]. (See also Section 3.2.2, Consistency.) Theoretically, it is possible to define the scope of Software Size measures precisely. However, the loss of generality that this introduces, limits their usefulness to Software Practitioners.

For Example:

> We return to our example (Page 9) of Blood Alcohol Concentration as a measure for Intoxication.

A person's Blood Alcohol Concentration is used as the legal measure of Intoxication for all people. (That is, its scope is all people.)

It is known that the presence of other drugs, such as Antihistamines, can affect a person's reaction to alcohol (their Intoxication). Thus, Blood Alcohol Concentration should only be used to compare the Intoxication of people who also have no Antihistamine - or the same level of Antihistamine - in their system.

Some criteria which should be considered when defining the scope of a Software Size measure are:

- application domain (Lokan, [Lokan, C., 1995], holds the opinion that Size measures should apply to all types of software.)
- programming language
- technologies
- reuse
- types of developments (new developments, enhancements to existing products and maintenance of existing systems)
- components of the Software which are included
  - Source Code, documentation and/or data (The cost of documenting software is high [Jones, C., 1994].)
  - Software, Software System, the Software Process, and/or the Software Development Process
- incomplete systems.

3.3.2 Calculation

Five factors are considered when looking at the Calculation criterion. These are: how easy it is to obtain the Size of a project - for New and Past Projects, and what support is available for determining the values of the measure (Software Support); when the Size can be obtained (First Phase Applicable) and what information is required (Database Required). It does not consider who can obtain values for the measure, or how the values should be obtained. This is discussed further at the start of Section 3.3.

Ease of Measuring or Calculating for New Projects

The values for some measures of Size are easier to calculate than others. Different Software Size measures require different information and have different approaches to combining that information. It may be easier to calculate the values for some Size measures than for others, either because the information they require is easier to

obtain, or because the method of combining information is easier [Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, 1990; Heemstra, F. J., 1992; Kusters, R. J. et al., 1990; Hufton, D. R., 1985]. The authors agree with Lokan, [Lokan, C., 1995], who says that the measures should be inexpensive to apply and calculate.

Ease of Obtaining for Past Projects

The ease of obtaining information for past projects also needs to be considered, [Hufton, D. R., 1985]. The information which is readily available for past projects is different to that which is readily available for new projects. This is particularly important in Software Costing where information from past projects is required to calibrate the Software Costing models.

Software Support

Automated (software) Support for Software Sizing approaches can significantly affect the ease with which Software can be Sized [Heemstra, 1990; Heemstra, F. J., 1992]. The automation may support the calculation of the measure from its sub-measures, and/or may derive the values for some, or all, of its sub-measures from other information.

First Phase Applicable

The first phase at which Software Sizes can be obtained is particularly important for Software Costing [Heemstra, 1990; Heemstra, F. J., 1992; Lokan, C. J., 1993; Lokan, C., 1995; Hufton, D. R., 1985]. Many measures can only be estimated early in the Software Development and cannot be calculated precisely until later in the Development.

Database Required

Some measures of Size are calculated or require calibration using a database of similar past projects.

If a database is necessary, then any organisations wishing to use that measure of Software Size need to 1) have developed similar project, and 2) kept the required information on them. This obviously limits the organisations which can use such Size measures.

3.3.3 Software Costing

When reviewing Software Sizing approaches for their use in Software Costing, we can consider how well the Size measure correlates with Effort. (Effort is considered to be the hardest to estimate, and most important, component of Software Cost [SPC, 1994].)

There are three areas (arising from statistics) which need to be considered when determining the suitability of Software Costing models:

- accuracy [Boehm, B. W., 1984; Boehm, B. W. and Wolverton, R. W., 1980; Heemstra, 1990; Heemstra, F. J., 1992; Kitchenham, B. A. and Taylor, N. R., 1984; Kusters, R. J. et al., 1990; Lokan, C. J., 1993; Lokan, C., 1995; Kemerer, 1991]
- precision [Kusters, R. J. et al., 1990; Lokan, C. J., 1993]
- statistically significant [Hufton, D. R., 1985]

A Software Costing model is Accurate if it is free of systematic biases. That is, the 'average' value of a complete set of valid estimates made using the equation is neither above nor below the 'average' of the true values. (Note that different types of 'averages' can be used.) The Precision of a model is a measure of the variation exhibited between the true, and the estimated, values. Standard Errors are a measure of Precision. Statistical Significance is usually determined by a statistical hypothesis test. These test the probability that the model occurred by chance.

4. Lines of Code - the Earliest Measures

4.1 Basis

Purpose

Lines of Code, or LOC, appears to be the earliest measure used for Software Size.

Historically, Size was important because of the memory limitations of early computers. The Size, measured in Lines of Code, was used to determine the memory required to store the program. The Lines of Code for higher level languages were converted to Equivalent Assembly LOC (EALOC) and the memory requirements determined from the EALOC.

Theory

This approach to determining memory requirements was soundly based on the theory, and knowledge of compiler construction. It used the fact that the initial compilers used tended to perform few, if any, optimisations.

Boehm proposed that the time (in staff-hours) taken to write software programs depended on their Size in Lines of Code, in a way that was independent of the language used [Boehm, B. W., 1984]. His opinion was based on observation and the belief that one Line of Code took the same amount of time to write regardless of the language. However, in his models, the Effort to write one line of code (and to provide associated documentation etc) depends on the type and Size of the application being developed. He did not provide an underlying theory to explain this phenomenon.

Class

Lines of Code is a measure of Length, as it considers 'how much' software there is rather than 'what' the software does (Functionality), or 'how' it does it (Complexity) [Lokan, C. J., 1993].

Source

Lines of Code is an Internal measure, as it is based on the Source Code of the system [Lokan, C. J., 1993].

4.2 Measurement

4.2.1 Repeatability

Clarity

The definition of Lines of Code given in [Conte, S. D. et al., 1986] is:

> *A line of code is any line of program text that is not a blank line or comment, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.*

One problem with this definition is that the value obtained depends on the style of the programmer. Programmers who write multiple statements per line would produce less Lines of Code than programmers who split statements across lines. Indeed, by passing the program though a simple style converter, the value of Lines of Code obtained could be radically changed.

Thus, while this definition is intuitively sound, it is also ambiguous. While this problem can be overcome by counting statements, or logical lines of code, rather than Lines of Code [USC, 1995], the definition of a statement is language dependent. The result is that Lines of Code becomes a less intuitively obvious measure, which is plagued by a plethora of language dependent definitions.

Even without allowing for variations between languages, several different definitions for Lines of Code have been used [Lesnoski, T. M., 1992]. He identified six major differences between Lines of Code counting methods (cited below), and the authors have identified other possible differences.

Most of these differences arise from the possibility of excluding certain types of lines. Examples are:

- blank lines
- comments (generally not included) [Lesnoski, T. M., 1992]
- Job Control Language (JCL) statements [Lesnoski, T. M., 1992]

- data (variable/constant) declarations [Lesnoski, T. M., 1992]
- format statements (eg "end") [Lesnoski, T. M., 1992]
- non-delivered code [Lesnoski, T. M., 1992].

Other reasons for differences in Lines of Code counts include:

- how reused code is counted [Lesnoski, T. M., 1992]
- how multiple statements on the same line are counted
- how the treatment of initialisation of arrays etc is counted
- how macros and procedures are counted
- the possibility of weighting certain statements, such as procedure calls.

Objectivity, Precision, Rater Independence

Given a rigorous definition, the process of determining the number of Lines of Code for a system can be automated, once the code is obtained, making the measure:

- objective
- repeatable
- capable of software support
- easy to use
- easy to obtain values for past projects (provided same language)
- absolute (certain).

However, many of the processes for obtaining early estimates of Lines of Code have none of these properties.

Known Information

Lines of Code cannot be measured prior to system coding. Methods for estimating Lines of Code are discussed in Section 4.3.2, however only a few of these use known information. The others rely on expert opinion or analogy.

Stability

Lines of Code is a stable measure as (individual) small changes to the system result in small changes to the value for the Lines of Code measure.

4.2.2 Soundness

Constructiveness

The authors believe that Lines of Code is not a constructive measure. It may be difficult to tell why one system is bigger than another. For example, two systems developed for the same purpose may have different Sizes. The reason for these differences cannot be determined from the Lines of Code measure alone.

Additivity

Lines of Code is an additive - and also an inverse additive - measure as the combined Size of a Software system is the sum of the Sizes of its components.

Minimality

There is no redundancy in the Lines of Code measure, as it has no sub-measures. However, there may be problems of redundancy when Lines of Code is estimated from other measures.

Completeness

It is difficult to determine if Lines of Code is a complete measure of Size for the purpose of Effort Estimation. Only the code component of Software is considered in Lines of Code and the authors do not know of any theory on how the code, documentation, and data components of Software are related. Measures of the Size of the documentation and data components may also be required, or may be highly correlated with the Size of code component. If the measures are highly correlated, then their inclusion would only lead to redundancy.

Consistency

Lines of Code cannot be consistently compared across different applications without considerable care. The Sizes of different Software Systems, measured in Lines of Code, can only be compared if the Systems were written in the same language. There may also be problems of consistency with application areas and design methodologies (see Section 3.2.1: Stability). Consistent use of Lines of Code may require these items to be factored out.

Furthermore, the same functionality may be delivered in different numbers of Lines of Code, particularly even if the same languages, developers, or design methodologies are used.

4.2.3 Other Measurement properties

Approach

Lines of Code can be measured directly once coding is complete.

Early Lines of Code estimates are often based on expert opinion or analogy although empirical estimation methods also exist (see Section 4.3.2).

Calibration

Lines of Code measures obtained directly from the source code do not require calibration.

However, approaches for estimating Lines of Code empirically do require calibration, at least for each language used.

Refinable

Lines of Code estimates can refined using the actual Lines of Code for the system. The ability to refine Lines of Code estimates during the development of the system depends on the estimation method(s) used. (See also 4.3.2.)

Scale

Lines of Code is an absolute scale. That is, it is a ratio scale and also a direct measure, or count.

Granularity

The granularity of Lines of Code will be used as a basis for comparison with other measures.

The granularity for estimates of Lines of Code would generally be finer than that for direct measurements of Lines of Code.

Decomposable

Lines of Code can be directly attributed to the components of a system, so the Lines of Code measure is decomposable.

Appropriateness

While Lines of Code, or EALOC are an appropriate measure of Size for determining if a system can reside in the memory of a computer, software engineers do not generally consider Lines of Code to be an appropriate measure of Size for Software Costing. The two main reasons for this are that:

- The Size of Software in Lines of Code is difficult to determine early in the software life-cycle [Kusters, R. J. et al., 1990].
- The Lines of Code measure is difficult to use consistently. (See Section 4.2.2: Consistency.)

4.3 Use

4.3.1 Scope

While it appears that the Scope of Lines of Code is generally large, the consistency (see Section 4.2.2) of Lines of Code measures is generally poor.

In [Nemecek, S. and Bemley, J., 1993] it is stated that AI systems contain no Lines of Code. However, the authors believe that there are equivalents to lines of code in AI systems. Furthermore, they believe that Lines of Code can be measured for all applications and domains, and that Lines of Code can be defined for most, if not all languages [Jones, C., 1995a].

Lines of Code can be used for the development of new systems, and the enhancement and maintenance of existing systems. However, the definition of Lines of Code tends to be adjusted to allow for differences between new, existing, modified and reused Lines of Code [USC, 1995]. These may either be used separately, or as a new measure, Equivalent New LOC (ENLOC) may be determined by weighting each of these Lines of Code sub-measures.

Lines of Code can also be used to measure the progress in developing a system, provided that the final Size of the system (in Lines of Code) can be determined. However, it can only be used to measure progress in the Software Coding, and not in the Documentation of the system, and provision of Data for the system.

4.3.2 Calculation

For Past Projects

The calculation of Lines of Code values for past projects is easy, provided that the source code is available. Automated procedures can be used to determine the number of Lines of Code directly from the source code.

For New Projects

For new projects, and projects currently under development, Lines of Code can only be obtained from the code after the Coding phase in complete.

Expert opinion, analogy, and empirical methods may be used to estimate the Lines of Code prior to coding. The precision and accuracy of these methods have undergone limited testing. However, using a small example and the PERT method, [Conte, S. D. et al., 1986] have shown that the average Size of systems tends to be underestimated and that the precision is very poor.

The PERT ([Lokan, C. J., 1993; Fairley, R. E., 1992]) and Statistical Sizing approaches, which acknowledge the distribution of possible Sizes ([Fairley, R. E., 1992]) are based on expert opinion. The PERT model is given in Equation 1.

$$LOC = \frac{Smallest\_Probable\_Size + 4 \times Most\_Likely\_Size + Largest\_Probable\_Size}{6}$$

Equation 1: PERT Sizing Technique

Successive Ratings, Pair-wise Comparisons and Paired Comparisons [Friedman, M. A. et al., 1995] are analogy based techniques. Each of these approaches involves comparing the Sizes of a collection of systems or modules.

In Successive Ratings, the length of modules are estimated by comparing them to the smallest and the largest modules. The smallest module is given a relative Size of 10 and the largest is given a relative Size of 100. The lengths of the remaining modules are determined from the average of their relative lengths compared to the longest and the shortest modules.

In Pair-wise Comparison, all the modules are compared on a Size basis to one module. A matrix is completed based on these values and an interval scale for the Sizes of the modules is determined by manipulation of the matrix.

In Paired Comparison, every pair of modules is compared in a random order. The results are placed in a matrix and it is manipulated to determine an interval scale for the Sizes of the modules.

Without going into details, empirically-based estimates of Lines of Code can also be obtained from:

- State machines [Britcher, R. M. and Gaffney, J. E., 1985]
- OO breakdown (function, data, interactions, non-functional) [Laranjeira, L. A., 1990]
- Data flow diagrams [Bourque, P. and Cote, V., 1991; Arifoglu, A., 1993]
- Structure charts and pseudo-code [Arifoglu, A., 1993]
- Function Points [Jones, C., 1995a; Jones, C., 1988; Verner, J. and Tate, G., 1988; Itakura, M. and Takayanagi, A., 1982; Reifer, D. J., 1988]. (See also Sections 6.2.3, 6.2.7, and 6.2.8).

Lines of Code cannot be estimated from the number of program variables [Levitin, A. V., 1987].

Software Support

Software Support is available for obtaining Lines of Code measures directly from the code. It is also possible to automate the process of obtaining Lines of Code estimates, although not the sub-measures on which they are based.

First Phase

Progressive measures of Lines of Code are available during the coding phase. Estimates may be available earlier, depending on the information on which they are based.

Database

Lines of Code measures obtained after coding is complete do not require a database of past projects. The estimation techniques, with the exception of PERT, which requires expert judgement, require a database of past, or standard projects for calibration.

4.3.3 Effort Estimation

The most commonly used model relating Effort to Lines of Code is COCOMO [Boehm, B. W., 1984; Boehm, B. W., 1988]. The COCOMO model has been widely published, unlike many other models relating Effort to Lines of Code, such as PRICE-S [Freiman, F. R. and Park, R. E., 1979]. The model is being updated to COCOMO 2 [USC, 1995].

The COCOMO model was initially developed and calibrated by hand-fitting the models to the data. Hand fitting data tends to result in models which apparently fit the data better than other approaches. However, it is likely that such models are actually over-fitted. That is, while the model fits the current data, it may not really capture the significant factors and so new data may not fit the model. Later work, such as [Kingston, G. et al., 1995; Jeffery, D. R. and Low, G., 1990; Kitchenham, B. A., 1992] has relied on statistical approaches to calibrate LOC-based Effort models. (Statistically-based approaches can also produce over-fitted models. However, if the approaches are applied correctly, and the number of factors which are incorporated in the models are limited, the likelihood of such problems is reduced.)

The main problem with using Lines of Code as an Effort estimator, is that Lines of Code measures are not available early in the development. The accuracy and precision of LOC-based models have not been tested using estimates of Lines of Code determined early in the development. Furthermore, the precision of Effort estimates made using the final Lines of Code appears to be poor [Kingston, G. et al., 1995].

Variants to the COCOMO model have been developed for particular languages and development environments, eg [Kaplan, H. T., 1991].

Lines of Code is also limited as a measure, or estimator, of productivity. Unless additional code or quality checks are performed, the Lines of Code counts can be manipulated by, for example:

- the inclusion of code which is not called
- using more lines than necessary to perform a simple operation, such as array initialisation.

Furthermore, if the Storage Size of the system must be limited, eg in an embedded system, the Effort required to reduce Size of the system to the required level is not captured [Kitzberger, B., 1995]. If the Size is being used as an Effort estimator, other factors may capture this apparent anomaly. If the Size is being used to measure productivity of the system's developers then it may be necessary to measure added, modified and deleted Lines of Code. However, if Size is being used to measure the Progress and estimate time to completion, Lines of Code alone is not an appropriate measure.

Lines of Code may be more useful in the estimation of Test or Maintenance Effort than for Development Effort. However, there are other code-based measures which

could be used to determine Test and Maintenance Effort, either in conjunction with, or as an alternative to Lines of Code.

5. Other Code-based Measures

Code-based measures may be grouped according to the Class of measures to which they belong. Complexity measures tend to be calculated algorithmically, while Code-based measures for the remaining classes of measures, Length and Functionality tend to be determined directly from Counts.

Complexity measures [Lederer, A. L. and Prasad, J., 1993a; Lederer, A. L. and Prasad, J., 1993b; Boehm, B. W. and Papaccio, P. N., 1988] arose from two sources:

- research into Software Testing, to determine test coverage, and
- research into maintenance, to determine how difficult it is to understand a software system.

Two of these measures will be discussed, Halstead's Software Science (Section 5.2) and McCabe's Cyclomatic Complexity (Section 5.3).

5.1 Counts

Some of the measures in this section, and some of those in Section 7 (on Other External Measures) can be determined from either the design or the code. That is, they could appear in either this section, or in Section 7. Those measures which appear in this section tend to have a value of more than one count per file, while those in Section 7 tend to have at a value of at most one count per source file. That is, the measures in this section can be used to measure the Size of a single Source Code file, and not just on the whole system.

The measures discussed in this section are:

- Classes and Methods per class [Fairley, R. E., 1992],
- Sub-programs [Freiman, F. R. and Park, R. E., 1979; Lederer, A. L. and Prasad, J., 1993a, Putnam, 1978] and
- Token counts. (Tokens are constructs, such as words or special symbols, which can be identified in the Source Code of a program.)

Code-based counts arose as potential improvements to the Lines of Code measure. Classes and Sub-programs provide higher-level abstractions to Lines of Code, which it may be possible to determine earlier than Lines of Code. Token counts were derived as a lower-level abstraction to Lines of Code to reduce the impact of programmer style and formatting on the value of the measure. Token counts are

often estimated by first estimating the number of Lines of Code, and multiplying by an average number of Tokens per line.

As these measures are similar to Lines of Code, only the differences will be discussed.

5.1.1 Background

Class

Classes and Sub-programs may be considered as measures of either Length or Functionality. As it is often assumed that Classes and Sub-programs may be programmed using an average number of Lines of Code (per Class or Sub-program), they may be considered a measure of Length. But, as they are independent of the language used, if not the type of language (procedural, functional, object-oriented etc), they may also be considered a measure of Functionality.

Token counts are a measure of Length, like Lines of Code.

First Phase Applicable

Token counts may only be measured after coding, while it should be possible to determine the number of Classes and Sub-programs from the detailed design as well as the code.

5.1.2 Measurement

Clarity

Classes and sub-programs are usually identifiable from the source code by identifying specific language constructs.

Token counts may be ambiguous in many of the same ways as Lines of Code counts. For example, tokens in comments may either be counted or excluded from the count. However, as with Lines of Code it is easy to define Token Counts unambiguously.

Granularity

The Sub-programs and Classes measures have a coarser granularity than Lines of Code, while Token Counts has a finer granularity.

Known Information

The number of Classes and sub-programs can be determined using known information once the detailed design of the system is known.

Token counts can only be measured - rather than estimated - after the system is coded.

Consistency

Sub-programs and Classes should be a more consistent measure of Software Size than Lines of Code when comparing systems developed in different languages. The Sub-programs and Classes measures depend of the detailed design of the system, and the final values may also depend on how the system is coded. The values from the code, and from the design may also be different. For example, if a function from the design is written "in-line" for efficiency reasons, the counts from the code and the design will be different.

The value of Token Counts should be more robust to style (particularly layout) differences between programs than Lines of Code. However, Token Counts depend on the system's implementation and problems still exist where tokens may be optionally included. For example, in Ada, procedure names may be optionally included at the end of procedures. Attempts to make Token Counts more robust in this respect reduce the ability to compare Token Counts between systems. As alternative definitions are introduced, it becomes necessary to discriminate between the variants.

5.1.3 Use

Scope

The Sub-programs and Classes measures should be less dependent on the development language than Lines Of Code. They will, however, depend on the type of language (procedural, functional, object-oriented etc), and the design.

Like Lines of Code, Token Counts will depend on the development language.

Costing

The authors believe that these measures *may* have been used in Software Costing models. However, they know of no sources in the open literature which document Software Costing models based on these measures.

5.2 Software Science [Halstead, M. H., 1977]

Halstead's Software Science proposes a method for estimating the Length of, and determining the Difficulty (or Complexity) of, software systems. It uses the following definitions and equations:

η_1, the number of unique operators

η_2, the number of unique operands

N_1, the total number of occurrences of operators

N_2, the total number of occurrences of operands

$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$, the Length of the system

$\eta = \eta_1 + \eta_2$, the Vocabulary of the system,

$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$, the Difficulty of the system, a measure of its complexity,

$V = N \log_2 \eta$, the Volume of the system in bytes, and

$E = DV$, the Development Effort in Elementary Mental Discriminations. This measure is commonly referred to as Halstead's complexity measure [Conte, S. D. et al., 1986; Friedman, M. A. et al., 1995].

5.2.1 Basis

Purpose

The Software Science measures were developed to provide a unified model for software programming. Several models of software programming had been developed before Halstead wrote his book on Software Science [Halstead, M. H., 1977]. His book was written to consolidate the thinking of the time and provide a basis for further work.

Class

Halstead's D and E (as defined above) may both be considered as Complexity measures.

Theory

Halstead's measures do not appear to have been based on any particular theory, and it has been shown that the Length measure in particular is invalid. In [Levitin, A. V., 1987], it is shown that no function of η_2 can be a good predictor of Software Size (as many systems - of differing length, functionality and complexity - can have the same value of η_2), while it has also been argued, [Fitsos, G. P., 1980], that η_1 should be approximately constant for large programs written in high level languages.

Source

The low-level counts for Halstead's Software Science can be derived from the source code, or estimated from the design. Halstead's Software Science provides equations for estimating the Length (in bytes) and determining a measure of Complexity from these low-level counts.

5.2.2 Measurement

5.2.2.1 Repeatability

Clarity

Halstead's original definition of operators was based on a program to implement Euclid's algorithm. Later work based on Software Science, [Fitsos, G. P., 1980], defines operators as:

> the operation codes, built in functions, delimiters, arithmetic symbols, punctuation etc. An operator then is an element which describes or implies how operands are to be used or operated upon. Macros and internal functions are also counted as unique operators

Both definitions of operators are ambiguous. For example, how many operators does '-' count for when it is both a binary and a unary operator which can be applied to two reals, two integers, two naturals, an integer and a real, or one integer? However, Halstead's definition does explain how to treat some similar situations. For example, paired operators, such as {} and begin/end, count as single operators. In many programming languages, these operators perform the same function, and therefore count as a single operator even if both are present.

Objectivity

Given a suitable definition, operators and operands can be objectively determined from the source code.

Known Information

The number of operands and operators is not known prior to software coding, which is when the Length equation would be most useful. However, it may be possible to estimate the number of operands from the detailed design, and according to [Fitsos, G. P., 1980], the number of operators is relatively constant for large programs written in high level languages.

Precision

There is no measure of uncertainty or precision for Halstead's Software Science Equations. (See also Section 5.2.1, Theory.)

Rater Independence

Prior to coding, the number of operators and operands needs to be estimated, so their values are likely to be Rater Dependent.

After coding is complete, the number of operators and operands can be automatically determined from the source code once a suitable definition of operators (or operands) has been developed.

Stability

The software science measures are stable as a small change in the number of operands or operators results in a small change in the values of the measures.

5.2.2.2 Soundness

Constructiveness

Halstead's Software Science measures are not constructive. The causes of differences in the complexities of two systems which result from differences in their values of N_1 or N_2 is difficult to determine.

Additivity

Both of Halstead's measures D and E are neither additive nor inverse additive. Using these measures, the complexity of a system may be either more (eg when all operators, but not all operands are used in each component) or less (eg when all operands, but not all operators are used in each component) than the sum of the measures of the components.

Minimality

It has been suggested that the number of operators in large systems developed in the same language is approximately constant. It therefore appears that this component could be replaced by a language constant [Fitsos, G. P., 1980]. There do not appear to be any other redundancy problems in Halstead's Software Science.

Consistency

The detailed counting rules for operators and operands are language dependent, [Fitsos, G. P., 1980], so the Length and Difficulty measures should not be compared for systems written in different languages.

5.2.2.3 Other Measurement Properties

Approach

Halstead's equations for D and E are algorithmic. The constants in the equations have not been justified theoretically or statistically verified.

The Length equation is empirical, in that it is theoretically proven (to determine the exact token count) and therefore does not require statistical validation.

Calibration

Halstead provides no mechanism for calibrating his approach.

Calibration is not required for the Length measure derived at the end of the project, as it is exact. Estimates of the Length could be made earlier in the development by a simple formula which multiplies the number of operators and operands by their average frequency in previous projects.

A calibration approach could not be derived for the complexity measures, D and E, until we have a better understanding of the attributes they are trying to measure.

Refinable

There are no methods for refining the measure other than refining an estimate to an actual value. Furthermore, the actual value for the Length could be determined more simply, without use of Halstead's formula, by simply counting the total number of Operators and Operands.

Scale

The components of Halstead's formulae are all directly measured, so the scale of all of Halstead's measures is Ratio.

Granularity

The granularity is the same as that of tokens, which is finer than that of Lines of Code.

Decomposable

Halstead's Software Science measures are not Decomposable. The measures may either be defined over the whole system, or over components of the system. However, the complexity of the system cannot be attributed solely to the complexity of its components.

Appropriateness

Several criticisms have been levelled at Halstead's Software Science [Levitin, A. V., 1987; Conte, S. D. et al.,]. These have addressed the time equation (not given in this paper) and variants of the length equation. They have not addressed the Complexity equations (D and E). However, these equations should be reviewed to determine if they are appropriate.

5.2.3 Use

5.2.3.1 Scope

The complexity equations should be appropriate for all types of new projects. It is not clear how they could be used to measure the Size of enhancements to existing systems.

5.2.3.2 Calculation

It may be possible to estimate Halstead's variables from the design of a system, but accurate values can only be determined from the source code. Once the source code is available it should be possible to automate the process of obtaining the variables and computing the desired measures.

As Halstead's measures do not require calibration, or comparison with past projects, no database is required.

5.2.3.3 Software Costing

Halstead's E equation is intended as a measure of the Effort to develop the software. Little work has been done to verify or refute this equation.

5.3 Cyclomatic Complexity [McCabe, T., 1976]

McCabe's Cyclomatic Complexity measure was developed as a measure of the number of basic paths through a program. The measure, called Cyclomatic Complexity, can be derived from a graphical representation of the program using the formula shown in Equation 2.

$$\text{Cyclomatic Complexity} = \#\text{Edges} - \#\text{Nodes} + 2$$

Equation 2: Cyclomatic Complexity.

5.3.1 Basis

Purpose

McCabe's Cyclomatic Complexity was proposed as a measure of how easy a system is to test and maintain. It is based on the number of paths through the software. These can be determined from the Control Flow Graph of the system.

Class

Complexity

Theory

Cyclomatic Complexity is based on Graph Theory.

Source

Cyclomatic Complexity may be considered either as an Internal measure determined from the source code, or as an External measure determined from the Control Flow Graph.

The properties of Cyclomatic Complexity depend on the source from which it is determined. When determined from the Source Code, the measure reflects the complexity of the delivered system. When determined from the Control Flow Graph, the measure is largely language independent.

5.3.2 Measurement

5.3.2.1 Repeatability

Clarity

Provided that a Control Flow Graph of the system exists (possibly embedded in the Source Code), the Cyclomatic Complexity of the system can be unambiguously determined. However, the procedure for determining an appropriate Control Flow Graph is not defined.

Objectivity, Known Information, Precision and Rater Independence

After the Control Flow Graphs are developed, the information required to calculate the Cyclomatic Complexity of a system (or component) can be determined directly from the graph. These graphs are often developed during detailed design. Cyclomatic Complexity cannot be determined before the Control Flow Graph is developed.

Stability

Cyclomatic Complexity is a stable measure. Because of the nature of Cyclomatic Complexity, the inclusion, or identification of additional non-branching code segments, have no effect on its value. The only changes which affect the measure are the addition, or removal of loops and branches. The addition of a single branch or loop increases the Cyclomatic Complexity of a system by 1.

5.3.2.2 Soundness

Constructiveness

Cyclomatic Complexity is not constructive. While increases in Cyclomatic Complexity can be easily traced to loops and branches, they cannot be easily traced to the requirements.

Additivity

Cyclomatic Complexity is inverse additive and not additive. If two components are added together sequentially, the complexity of the resulting system is the sum of the complexities of the two systems, minus one. However, if a component is added as a procedure call, then the complexity of the resultant system is the sum of the complexities of the two components.

Minimality

Cyclomatic Complexity is Minimal. The number of edges and components in a Control Flow Graph tend to be highly correlated. However, only the difference between these numbers is included in the model.

Completeness

From Graph Theory it can be shown that Cyclomatic Complexity is complete.

Consistency

The Cyclomatic Complexity of different systems can be consistently compared regardless of the application, or the technology used to develop the system. However, different technologies may result in systems of differing Cyclomatic Complexities (systems with different Control Flow Graphs) being developed for the same requirements.

5.3.2.3 Other Measurement Properties

McCabe's Cyclomatic Complexity was derived using an algorithmic approach. It is based on Graph Theory and not on calibrated information. The justification for its use as a measure of complexity is incomplete. In particular, the +2 in the equation is not justified empirically and its theoretical justification is weak.

It cannot be calibrated or refined.

Cyclomatic Complexity is an ordinal scale measure with a coarser granularity than Lines of Code.

The complexity of the system can be attributed to the complexity of its components, so it is decomposable.

It does not appear that Cyclomatic Complexity could be used in isolation to predict development Effort. It is, however, definitely appropriate for determining the number of paths through a software system.

5.3.3 Use

5.3.3.1 Scope

Cyclomatic Complexity can, in principal, be determined for any system. However, some development languages (eg some fourth and fifth generation languages) and support tools hide the flow of control from the developer. It may be harder to determine the Cyclomatic Complexity for systems developed using these approaches.

Cyclomatic Complexity may be a useful measure during the maintenance of software systems.

It is not likely to be a useful measure of Software Size for Effort estimation for either new developments, or enhancements. This is because systems with little control flow will all have similar Cyclomatic Complexities regardless of their Functionality or Length.

5.3.3.2 Calculation

Cyclomatic Complexity can be determined for any system for which a Control Flow Graph exists, which is often the case once the detailed system design has been developed.

The Cyclomatic Complexity can also be determined automatically from source code for a system, provided all language constructs which indicate branching of control can be identified.

It may be possible to develop a tool to support the computation of Cyclomatic Complexity from the source code which is language independent. This would require a database (of language constructs which indicate branching) to be developed for each language.

5.3.3.3 Software Costing

The authors know of no work in the open literature which relates Cyclomatic Complexity to development Effort, or Cost. However, as a very long, sequential program has the same complexity as a single line program, it is likely that Cyclomatic Complexity alone would not correlate well with Effort.

6. Function Point Measures

Functionality measures are an alternative class of Size measure whose values are generally obtainable earlier in the Software development than Complexity or Length measures.

The work of [Brooks, F. P., 1987] provides motivation for considering these types of Size measures. He states that:

> *The essence of a software entity is a construct of interlocking concepts: data sets, relationships between data items, algorithms, and invocation of functions.*

6.1 Albrecht FP

One of the earliest, and best known, measures of functionality is Albrecht's Function Points [Albrecht, A. J. and Gaffney, J. E., 1983; Morris, P., 1994]. The International Function Point Users Group (IFPUG) is working towards creating an International Function Point Standard based on this measure of Software Size. An alternative version of Function Points, MkII (see Section 6.2), also has wide recognition, particularly in Europe.

Albrecht Function Points (AFP) are defined as the product of a Raw Function Point Count with an Adjustment Factor.

The Raw count is a sum of adjusted function counts. Five types of functions are considered: Inputs, Outputs, Enquiries, Internal Logical Files and External Interface Files. Their counts are adjusted according the complexity of the functions.

Table 1: Function Point Technology Factors

1.	Data Communications
2.	Distributed Data Processing
3.	Performance
4.	Configuration Usage
5.	FP -Transaction Rates
6.	On-line Data Entry
7.	End-User Efficiency
8.	On-line Update
9.	Complex Processing
10.	Re-useability
11.	Installation Ease
12.	Operational Ease
13.	Multiple Sites
14.	Facilitate Change

The Adjustment Factor ranges in value from 0.65 to 1.35 and is obtained using 14 Technology Factors shown in Table 1. (Note that according to [Abran, A. and Robillard, P. N., 1994], Albrecht's original definition contained only one File type, only 10 factors (Factors 1 and 11-14 replaced the single original factor Backup), and allowed no adjustment for the complexity of functions.)

6.1.1 Basis

Purpose

The purpose of Albrecht's Function Points was to provide a measure of system functionality which viewed the system from the users' perspective [Albrecht, A. J. and Gaffney, J. E., 1983].

Class

Functionality.

Theory

Function Points were determined by a process of trial and error, rather than from a theoretical basis.

An attempt was made to formalise the basis for Function Points by comparing Function Points to Halstead's Software Science [Albrecht, A. J. and Gaffney, J. E., 1983]. This showed that there was a form of equivalence between the two measures.

Using a set of 24 systems this study also showed that Function Points correlated well with Lines Of Code, and with Development Effort. However, the work of Levitin [Levitin, A. V., 1987], shows that Software Science (see Section 5.2) is not a good estimator of Lines Of Code. If, as stated above, there is some form of equivalence between the two measures, then these results are surprising. The authors believe that this may be because Albrecht's results came from a particular development environment, and they believe that, given the work by Levitin, it is unlikely that they would hold in general.

As the theory behind Halstead's Software Size is limited, equivalence with it does not provide a theoretical basis either for using Function Points as a measure of system functionality from a user's perspective, or for estimating Effort.

Source

External.

6.1.2 Measurement

6.1.2.1 Repeatability

Clarity

The definition of Function Points is not Clear. The original definition of Function Points provided a high level description with an intuitive meaning. However, the components are difficult to describe further and this leads to ambiguity in the definition. Even the current IFPUG definition is ambiguous, although it is being refined continually.

Objectivity

Function Points are not an objective measure of Software Size. In particular, the distinction between Outputs and Enquiries is not clear. Guidelines do exist for determining whether a function is an Output or an Enquiry [Morris, P., 1994], but the authors feel that even these require some level of subjectivity. The authors believe that most Function Point proponents think that, with appropriate training, there is little difference between the values obtained by different raters. (See Rater Independence.)

Known Information

The required information can be determined once a functional decomposition has been performed, the data requirements for each function determined, and the Technology Factors determined. This information is normally available on completion of the Design phase, and may be known even earlier.

Precision

Attempts have been made to quantify the uncertainty, or subjectivity, in Function Point estimates [Yau, C. and Tsoi, R. H. L., 1994]. These are limited to the assessment of the Technology Factors. The approach only allows for the uncertainty known by the rater to be captured, and does not allow for any inherent uncertainty to be modelled or captured. (This could include uncertainty inherent in the model, or in the rater due to their inexperience.)

Rater Independence

It was stated in [Banker, R. D. et al., 1994] that, even when Function Point values are obtained by experienced raters, differences occur which require resolution. In a statistical study, [Kemerer, C. F., 1993], it was concluded that the correlation between values obtained by different raters was high, and that the values obtained were not statistically different. However, the percentage difference between the values obtained by the different raters had a median of 12% and a mean of 27% indicating a moderate amount of imprecision.

Stability

Function Point counts are generally fairly stable. Small variations in the values used to determine Function Points tend to result in small variations in the overall Size (in Function Points). The Technology Factors could have a big influence on the Size of large systems. However, when they are combined into the Adjustment Factor, their effect tends to be a multiplier of around 1. That is, they tend to have little net effect on the value obtained for the Size of the System. Function Points appear to be relatively stable, but it appears that variations do not always cancel out and large variations between Function Point counts can occur.

6.1.2.2 Soundness

Constructiveness

The source of differences in the "Sizes" of systems, measured in Function Points, can be clearly identified. Therefore, the measure is Constructive.

Additivity

Function points are inverse additive and not additive [Symons, C. R., 1988]. Because of the way interfaces are counted, the Function Point count of an integrated system is at most the sum of the Function Points of its component systems.

Minimality

Several redundancy problems have been noted with Function Point measures.

Correlations between the sub-measures [Stathis, J. and Jeffery, D. R., 1993; Kitchenham, B. and Kansala, 1993] have been noted. (These studies resulted in different correlations, and the cause of these differences is not known.)

Complexity is accounted for twice: once in the weighting of the functions, and once by the Technology Factor, Complex Processing [Symons, C. R., 1988]. It is possible that these could be defined so there is no overlap between the two, but this is yet to be done. An alternative is to drop one of the measures of complexity.

The results of other researchers, [Kingston, G. et al., 1995; Kemerer, C. F., 1987; Jeffery, D. R. et al., 1993], indicate that the Adjustment Factor and most of the Technology Factors do not have a statistically significant effect on Effort estimation. It appears that most of the Technology Factors, including Complexity, could be ignored. IFPUG currently plan to drop the Adjustment Factor from the definition of Function Points.

Completeness

It is commonly accepted that Function Points are not appropriate for all types of systems (see 6.1.3.1). In addition, Function Points do not capture all types of functions (see the comments on Mark II Function Points in Section 6.2.1.1).

Consistency

Function Points appear to provide a consistent measure of functionality for those systems which fall within its scope. The measure is independent of the language used, and appears to be independent of the development approach (eg object-oriented or structural) used.

6.1.2.3 Other Measurement Properties

Approach

The Function Point measure is algorithmic. Its weights, and the influence of its technology factors, were determined by expert opinion. It has been suggested that it should be calibrated [Verner, J. and Tate, G., 1992], which would make it empirical. This approach was used with Mark II Function Points [Symons, C. R., 1988].

Calibration

Albrecht's proposal did not allow for calibration of the approach. The weights applied to the functions, and the impact of the Technology factors were determined using trial and error. However, as it is believed that these results may be environment specific, it is suggested that calibration is required for other environments [Verner, J. and Tate, G., 1992].

Refinable

While a precise value for Function Points may not be available before the completion of the Design phase, Function Points may be estimated with increasing precision using information available very early in the Software Development [Arifoglu, A., 1993].

Further refinement is not possible after Design (although changes to the functionality required may change the Function Points of the system). Conversion to Lines of Code does not help, as Function Points do not map well to Lines of Code at the module level [Verner, J. and Tate, G., 1992].

Scale

While Function Points are often treated as a ratio measure, [Abran, A. and Robillard, P. N., 1994] have argued that Function Points are an index not a measure. The transformations used in the calculation of Function Points transform the original counts (which are of ratio scale) into results of nominal scale. Without an underlying theory, (such as correlation for each Function Type with Effort [Symons, C. R., 1988]), the final transforms are invalid, and the Function Point value obtained should not be treated as a ratio scale.

Granularity

Function points is a coarser measure of Software Size than Lines Of Code, but has a similar granularity to the other Code-based measures (excluding tokens).

Decomposable

The Function Point measure is not decomposable. Although Function Points appear to capture system functionality at a high level, it has been shown that Function Points do not capture functionality well at the module level [Verner, J. and Tate, G., 1992; Shepperd, M., 1992].

Appropriateness

The appropriateness, or otherwise, of Raw Function Point counts depends on the appropriateness of the function types counted, which is difficult to determine, and the appropriateness of their combination (see Scale).

The appropriateness, or otherwise, of the Adjustment Factors depends on the appropriateness of both the Technology Factors, and how the Technology Factors are combined. None of these appear to be appropriate as the Technology Factors have been shown to be dependent on time [Verner, J. M. et al., 1989], and to have different effects from each other [Symons, C. R., 1988].

6.1.3 Use

6.1.3.1 Scope

The scope of Function Points is limited to transaction-based systems because Function Points do not attempt to capture processing internal to the system [Jones, C., 1995b]. These are typically business information systems [Lokan, C. J., 1993]. This means that Function Points do not capture well the Size of real-time [Jones, C., 1995b; Reifer, D. J., 1988], scientific [Reifer, D. J., 1988] or, embedded, communications, process control or systems software [Jones, C., 1995b].

Function Points appear to be language independent although variants have been used for developments in particular types of languages. These include variants for

developments in Object Oriented languages [Banker, R. D. et al., 1994], and variants for developments in 4GLs [Verner, J. and Tate, G., 1988]. A table giving the average number of Lines of Code per Function Point for a variety of programming languages has been published [Jones, C., 1995a].

Function Points were developed to be technology independent. However, the Technology Factors, and the types of functions expected by the user, are technology dependent [Verner, J. M. et al., 1989].

There are methods for obtaining the number of Function Points for project enhancements [Morris, P., 1994], handling reuse and maintenance of projects [Banker, R. D. et al., 1994] as well as for new projects. These methods appear to work well [Abran, A. and Robillard, P. N., 1993; Kingston, G. et al., 1995].

6.1.3.2 Calculation

Ease of Measuring or Calculating for New Projects

Imprecise values of a system's Function Points can easily be obtained early in the system's development. (See also Section 6.1.2.4, Refinement).

Ease of Obtaining for Past Projects

Function Points may be easily obtained for past projects on which the appropriate information was collected and retained. If the functional decomposition and documentation of the complexity and Technology Factors are provided, Function Point counts can be calculated for past projects as easily as for new projects.

It is difficult to obtain Function Point counts for projects where the required information was not originally collected. While the authors have heard that the Function Point counts can be obtained from the source code, they believe that this is not generally possible. Even where this may be possible, biases can be introduced. For example, by considering the implementation (screens etc) used rather than knowledge of the (original) functional requirements.

Software Support

Several software support tools are available [Charismatek, 1994; Matson, J. E. and Mellichamp, J. M., 1993; Banker, R. D. et al., 1994]. Some of these simply support the calculation and storage process while others simplify the calculation process - for example, by assuming average complexity for all functions - and determine the functions from design information.

First Phase Applicable

Opinions vary on when Function Points can first be calculated. The most common opinion is that Function Points can be estimated from the Specification and computed precisely during, or after, the Design Phase [Arifoglu, A., 1993; Mukhopadhyay, T. and Kekre, S., 1992].

Database Required

As it is recommended that Function Points be calibrated [Verner, J. and Tate, G., 1992], a database is required. It should be large enough to allow the calibration to be statistically significant.

6.1.3.3 Software Costing

Models have been developed for explaining the relationship between Effort and Function Points [Albrecht, A. J. and Gaffney, J. E., 1983; Kingston, G. et al., 1995; Matson, J. E. et al., 1994]. Statistical studies have shown that these types of models perform best when the impact of the local environment is recognised, either through calibration [Jeffery, D. R. and Low, G., 1990], or by identifying the data sources through a factor in the models [Kingston, G. et al., 1995].

Statistical studies have shown that the Function Point Adjustment Factor has little impact on the correlation of Function Points with Effort [Kitchenham, B. A., 1992].

Other statistical studies [Kingston, G. et al., 1995] have shown that the precision of estimates made using Function Points is less than that for Lines of Code, although the upper bounds of Prediction Intervals for Function Point and Lines of Code based models are similar. However, Function Point estimates may be more precise early in the software development as, unlike for Lines of Code, Function Points do not need to be estimated early in the development. The imprecision in the estimates of Lines of Code may increase the imprecision of Lines of Code-based estimates so that it is greater than for Function Point-based estimates.

Function Points are better than Lines of Code for comparing the productivity of different projects and development methods [ASMA, 1994; Jones, C., 1995b]. This is because productivity improvements may reduce the number of Lines of Code developed, while the number of Function Points remains unchanged.

They may also be useful for estimating the Cost of developing documentation for the user, as it is a measure of functionality of the system from the user's perspective.

6.2 Variants

Several variants of Function Points have been developed to overcome some of the problems of Albrecht's Function Points. Some of these address technical issues, while others are aimed at broadening the applicability of Function Points.

The evaluation criteria are used to discuss the differences between these measures and Albrecht's Function Points. Problems common to several variants are discussed under the variant with which they are associated in the literature. The Basis and some of the other criteria are omitted where the information follows from that of Albrecht's Function Points and/or there is insufficient additional information in the literature to warrant their inclusion.

The variants are identified by a name and a reference. Unless otherwise stated, statements about the variant are derived from this reference. Where the variants are not named in the reference, names are provided by the authors and the reference is given in italics.

6.2.1 Mark II Function Points [Symons, C. R., 1988]

Symons developed Mark II Function Points to address some of the concerns he had with Albrecht's Function Points. Mark II Function Points are similar to Albrecht's, except that:

- Enquiries are treated as a combination of Inputs and Outputs.
- Logical Internal Files and External Interface Files are reclassified as Entities.
- The weights given to the different types of functions were determined by calibration against Effort. (Note though, that the Effort breakdown used by Symons was determined subjectively.) This makes the approach empirical, rather than algorithmic.
- Functions which cross applications are counted for each system.
- Additional Technology Factors are allowed and those suggested include:
    * Software Interfaces
    * Security
    * Third Party Access
    * Documentation
    * User Training.

6.2.1.1 Soundness

Additivity

The perceived Additivity problems of the Function Point measure were addressed by considering the application, rather than the system boundary [Symons, C. R., 1988]. Mark II Function Points are both additive and inverse additive.

Consistency

Consistency problems are introduced by not standardising the Technology Factors and the weights of the functions.

Mark II Function Points cannot be compared without first ensuring that the same factors were used. If different factors were used, the comparisons may be invalid. It is suggested, [Whitmire, S. A., 1992], that a new Adjustment Factor should have been proposed.

Determining the weights of the functions from the Effort makes them technology dependent [Hughes, R., 1994; Whitmire, S. A., 1992]. Furthermore they were derived from estimates [Symons, C. R., 1988; Whitmire, S. A., 1992].

Completeness

In a critique of Mark II Function Points, [Hughes, R., 1994], several problems were identified in the completeness of Function Points:

- Control elements, such as menus, are not counted.
- Housekeeping, such as backups, are not counted.
- The treatment of dates, eg a single year, or a triplet of values, is not addressed.
- The treatment of variants of reports, eg graph versus table or different sorting orders, is not addressed.

These critiques also apply to Albrecht's Function Points, and other Function Point variants.

6.2.1.2 Other Measurement Properties.

Calibration

Calibration is required for Mark II Function Points.

Scale

Mark II Function Points are also treated as a ratio scale. There is some justification for this scale as the weights of the functions are determined on the basis of the functions' correlation with Effort. However, Mark II Function Points should still be treated warily as there is no explanation for why the weights for each type of function are constant in each environment, but vary between different environments. That is, the cause of the variation is not identified.

6.2.1.3 Calculation

Database

A database is required for determining Mark II Function Points as the model must be calibrated for each environment. The data required is not commonly collected [Symons, C. R., 1988].

6.2.1.4 Costing

A test of Mark II Function Points was performed by using the Before You Leap tool to predict the Cost of systems which were not used to calibrate the tool [Betteridge, R., 1992]. It was found that there was a bias in the estimates of 5% (that is, the estimates were not accurate) and that the estimates average absolute error (a measure of the precision) was 20%. Note however, that the bias in the estimates is not surprising given that Symons intended Mark II Function Points to be calibrated to the local environment.

6.2.2 SPQR

SPQR Function Points were developed by Capers Jones to incorporate suggested improvements to Albrecht's Function Points [Jeffery, D. R. et al., 1993]. SPQR Function Points are similar to Albrecht's, except that:

- All functions are assumed to be of average complexity.
- Only two Technology Factors, Logical Complexity and Data Complexity, are used. (Note that this approach differs from that used for Mark II Function Points [Symons, C. R., 1988] where more, rather than less, Technology Factors are used.)

6.2.2.1 Repeatability

As all functions are assumed to be of average complexity, the complexity of the functions does not need to be determined. The means that

- one cause of rater dependence (differences in the complexity) is removed and
- less sources of information (about the complexity) are required.

However, the assessment of the two Technology Factors on complexity may re-introduce rater dependence. In [Jeffery, D. R. et al., 1993] the approach to determining these factors was to sum the values of the Albrecht Technology Factors which fell into each category. This suggests that the SPQR Technology Factors are not well defined.

6.2.3 ASSET-R Function Points [Reifer, D. J., 1988]

ASSET-R Function Points were developed to extend the scope of Function Points to scientific problems and provide an appropriate relationship between Function Points and Lines of Code.

ASSET-R Function Points are like Albrecht's Function Points except that they include the additional functions:

- modes
- rendezvous (a synchronisation mechanism) and
- stimulus/response pairs.

6.2.3.1 Repeatability

Clarity

No definition of modes, rendezvous or stimulus/response pairs is given in [Reifer, D. J., 1988; Whitmire, S. A., 1992]. Therefore they are not clearly defined. The remaining functions, which are derived from Albrecht's definition of Function

Points, are defined; but no less ambiguously than in the original Albrecht definition. ASSET-R Function Points are therefore less clearly defined than Albrecht Function Points.

6.2.3.2 Soundness

Minimality and Consistency

Without detailed definitions of the functions it is difficult to determine if consistency problems, or additional redundancy problems have been introduced.

6.2.3.3 Other Measurement Properties

Appropriateness

The number of rendezvous is implementation dependent, so the value of ASSERT-R Function Points may vary with the low-level design [Whitmire, S. A., 1992].

6.2.3.4 Scope

ASSET-R Function Points claim to extend the scope of Function Points to scientific problems.

6.2.3.5 Costing

No methods exist for determining the Cost, or the Effort, required to develop a software system using ASSET-R Function Points.

Instead Equation 3 is provided to convert ASSET-R Function Points (FP) to Lines of Code (LOC) which can presumably then be used to estimate Effort. To the best of the authors' knowledge this equation has not been validated statistically.

$$LOC = (ARCH)(EXPF)((LANG \times FP) + MV)^{REUSE}$$

Equation 3: ASSET-R Function Point to Lines of Code Conversion

The variables in the equation refer to the ARCHitecture of the system, the EXPansion Factor, the LANGuage expansion factor, the Maths Volume and the REUSE factor. The expansion factor captures differences in:

- Language Experience
- Environment Experience
- Database Size
- Programming Techniques
- Software Tools
- Analyst Capabilities
- Applications Experience

- Requirements Volatility
- Real-Time Requirements.

6.2.4 Feature points [Jones, C., 1995b]

Jones developed Feature Points to extend the scope of Albrecht's Function Points. Feature Points are similar to Albrecht's, except that:

- An additional component, Algorithms is included. Jones defines an algorithm as "the set of rules which must be completely expressed to solve a significant computational problem" and as "a bounded computational problem which is included within a computer program". He gives examples of square root extraction and Julian date conversion routines.
- The weight of algorithms can vary from 1 to 10 depending on their complexity. The default weight is 3.
- The weight of Logical Files (Logical Internal Files and External Interface Files) is lower.

6.2.4.1 Repeatability

Clarity

While Jones further discusses how to identify algorithms, the definition is ambiguous. The problem of how to determine the appropriate weight for the algorithm is even more ambiguous.

Known Information

The number, and the complexity, of the algorithms used by a system may not be known as early in the system development as the other Function Point components. Jones recognises this and says that Function Points should be used in these circumstances.

6.2.4.2 Soundness

Minimality

Feature Points are not Minimal. The use of the Technology Factor, Complex Processing, and the algorithms component in the same model means that complexity is counted twice [Whitmire, S. A., 1992].

Completeness

Feature Points are not Complete. They do not capture control aspects, nor do they account for all aspects of internal processing [Whitmire, S. A., 1992].

6.2.4.3 Other Measurement Properties

Refinement

Feature Points can be refined from estimates based on Function Points once details of the algorithms are known. Jones provides a table of the standard ratios between Function Points and Feature Points for different types of systems. This could be used to estimate the number of Feature Points prior to the availability of information about the system's algorithms. The estimate can be refined once details of the algorithms are known.

Appropriateness

Although Jones states that Feature Points are experimental, preliminary results indicate that Feature Points are more appropriate than Function Points for scientific and engineering applications.

6.2.4.4 Scope

Feature Points appear to be applicable to a wider class of systems than Function Points. It has been applied to several kinds of systems including: Transaction-based systems, PBX Switching systems, Embedded Real-Time systems and Factory Automation systems [Jones, C., 1995b].

It is possible that Feature Points may provide a better measure of progress in development than Function Points because they measure more of the detailed processing required in the source code. However, there is no evidence to support this conjecture.

6.2.4.5 Calculation

First Phase Applicable

Feature points cannot be calculated before the algorithms have been determined. All the algorithms to be used are not likely to be known before, at least, detailed design.

Software Support

The CheckPoint tool supports the calculation of Feature Points [Jones, C., 1995b].

6.2.5 3-D FP [Whitmire, S. A., 1992]

One of the more elaborate extensions to Function Points is 3-D Function Points [Whitmire, S. A., 1992]. Whitmire proposes that Software has three dimensions, Data, Function and Control. He claims that Albrecht Function Points only capture the Data Dimension and proposes additional measures to cover the Function and Control dimension.

The Function dimension is measured by:

- Counting the number of operations which transform information. Formatting and ordering of the data are not counted.

- Each operation is weighted according to its complexity which is determined from:
  - the number of semantic statements, which is the number of pre-conditions, post conditions and invariants counted for each transform, and
  - the number of process steps.

The Control dimension is measured from a finite state machine representation, such as a State Transition Diagrams by:

- Counting the number of states
- Counting the number of actions which can cause a state change.

6.2.5.1 Repeatability

Clarity

The definition of 3-D Function Points is not clear. It does not describe how to determine the number of process steps of a transaction, or how to determine states and actions for representation in a state transition diagram.

6.2.5.2 Soundness

Minimality

3-D Function Points may not be Minimal. It is not clear from Whitmire's description that the Data, Function and Control dimensions of 3-D Function Points are independent. In particular, it would seem possible that many transformations could be counted twice, once in the Function and once in the Control dimension.

6.2.5.3 Scope

3-D Function Points appear to be applicable for a much wider class of systems than Albrecht's Function Points. It is not clear whether 3-D Function Points or Feature Points are more appropriate for the various domains covered by this wider class of systems. 3-D Function Points consider more views of the software. However, it is not clear whether or not some of these views are redundant.

6.2.6 Data Points [Sneed, H., 1994]

Sneed has proposed Data Points as an alternative to Function Points. It was Sneed's intention that, for Object-Oriented Developments, Data Points would be applicable earlier than Function Points . As the authors believe that this is yet to be published, it is discussed only briefly.

Data points are obtained from an Entity Relationship breakdown of the system, rather than a Functional breakdown. Equation 4 depicts the Data Points model which consists of an Information and a Communication component. It appears to

suffer from many of the same problems as Function Points, although there may be less redundancy problems.

$$\sum_{\text{Entities}}(\text{Data Fields}+2\times\text{Keys}+4\times\text{Relations}+4)+\sum_{\text{Interface Objects}}(\text{Data Fields}+2\times\text{Keys}+4)\times\begin{Bmatrix}1.2\ \text{Input}\\1\ \text{Other}\end{Bmatrix}$$

Equation 4: The Equation for Calculating Data Points

An alternative measure based on Entity-Relationship diagrams was analysed in [Kemerer, C. F., 1987] and Object Points, another measure for Object-Oriented Systems which is more closely related to Albrecht's Function Points, is discussed in [Schooneveldt, M. et al., 1995].

6.2.7 Interface Points *[Verner, J. and Tate, G., 1992]*

In 1992 Verner and Tate proposed a measure of Software Size similar to that of Function Points except that:

- The components counted depend on the development technology. For the 4GL, Application Language Liberator or ALL, they consider they proposed Menus, Relations, Screens, Reports and Updates as the components as these were handled differently by ALL.
- The weights of the components are determined by the average number of lines of code for each type of component.

Because this measure focuses on the user interface to the system, we have called it Interface Points rather than Function Points.

Interface Points address the problem which Function Points have determining the Effort required to develop individual modules. However, it introduces (additional) technology dependence so Interface Points for different systems cannot necessarily be compared.

A lower level measure, specifically designed to measure the Size of Graphical User Interfaces is described in [Lo, R. et al., 1995].

6.2.8 Banking Points *[Itakura, M. and Takayanagi, A., 1982]*

In 1982 Itakura and Takayanagi proposed a Function Point like measure for estimating the Size of Banking Systems in Lines of Code. They noted that they required different models for Cobol and Assembly programs, and that different models were required for different types of Assembly programs.

The main model they proposed was their Cobol model, which is similar to Function Points except that:

- Input files, input items, output files, output items, reports, horizontal items in reports, vertical items in reports, calculating processes, sorting and report types were considered as potential factors in their model. Sorting was found to have no effect, so was ignored in the model. Calculating processes were assumed to have 80 steps each.

- There are no Technology Factors.

Itakura and Takayanagi considered how the components could be combined to reduce the complexity of the equations. This reduces the number of variables in the equations and allows the equations to be calibrated with less data. Using this method they derived a more precise Effort estimation equation.

While Banking Points have a better foundation that Function Points, most of the problems with Function Points still apply to Banking Points.

6.2.9 Other

There are several other variants of Function Points, but all have similar problems to the Function Point measures discussed in Sections 6.1 to 6.2.8. It is clear from a quick comparison of Function Point measures that there is no consensus in the Software Engineering Community regarding the desirable properties of a measure of Functionality.

Some of the measures increase the complexity of Function Points. For example, AI Function Points, which allow a domain component, are discussed in [Nemecek, S. and Bemley, J., 1993]. Fuzzified Function Points [Yau, C. and Tsoi, R. H. L., 1994] allow fuzzy values for the subjective Technology Factors.

Other measures decrease the complexity of Function Point Counts. A simplified version of Function Points, Unweighted Function Counts, is documented in [SPC, 1994]. In Unweighted Function Counts, complexity factors are not applied to the components.

6.3 Comparisons

Comparisons have been made between Function Point measures to determine if the measures are equivalent, or to determine which are the best measures for estimating Effort.

Studies have shown that Albrecht Function Points have a high correlation with both Entity Relationship Function Points [Kemerer, C. F., 1987] and DeMarco's Function Bangs [Rask, R. et al., 1993] although neither of these studies considers how any of the Function Point measures relate to Lines of Code, or to Effort.

A more in-depth study, [Jeffery, D. R. et al., 1993] comparing SPQR to Albrecht Function Points showed that there is little difference between the measures for Effort Estimation and in their overall counts. However, it showed that there were still

statistically significant differences between the two Function Point measures. SPQR tended to provide higher counts for Unadjusted Function Points than Albrecht's values, the Adjustment Factors were different, and the variance of the counts was different. This means that while there was a good correlation between the two measures, that they are not interchangeable.

Another study comparing the ability of different Function Point models to estimate Effort is described in [Ferens, D. V. and Gurner, R. B., 1992]. This model considered three Effort Estimation Tools: SPANS, which uses Albrecht Function Points; CheckPoint, which uses Albrecht Function Points or Feature Points; and Costar which uses a Function Point variant. When calibrated for the data, the results for the three models were similar. When not calibrated, all the models tended to overestimate Effort, with CheckPoint providing the most accurate estimates, and Costar providing the least accurate estimates. The precision of the models was considered poor with no model predicting the Effort to within 30% of the actual Effort more than 50% of the time.

Table 2: Components of Selected Function Point Measures

	Albrecht	FP Mark II	Feature Points	SPQR-20	ASSET-R FP	3D-FP	Data Points	Interface Points
Input	X	X	X	X	X	X		
Output	X	X	X	X	X	X		
Enquiry	X		X	X	X	X		
External File	X		X	X		X		
Internal File	X		X	X	X	X		
Entity		X						
Relations							X	X
Data Field							X	X
States						X		
Transaction						X		
Keys							X	
Mode					X			
Stimulus/Response					X			
Rendezvous					X			
Algorithms			X					
Transformations						X		
Menu								X
Screen								X
Report								X
Updates								X
Line Types								X
Control Breaks								X
Nesting Level								X

Some of the differences between the components and Technology Factors used by the different Function Point measures are summarised in Table 2 Table 3. Other differences between Function Point variants are discussed in [Hastings, T., 1995].

Table 3: Technology Factors for Selected Function Point Measures and Related Tools

	Albrecht	FP Mark II	Feature Points	SPQR-20	ASSET-R	3D-FP	Data Points	Interface Points
Data Communications	X	X				X		
Distributed Functions	X	X				X		
Performance	X	X				X	X	
Heavily Used Configuration	X	X				X		
Transaction Rate	X	X				X		
On-Line Data Entry	X	X				X		
End User Efficiency	X	X				X	X	
On-Line Update	X	X				X		
Logical Complexity				X				
Data Complexity				X				
Complex Processing	X	X				X		X
Re-Usability	X	X				X		
Installation Ease	X	X				X		
Operational Ease	X	X				X		
Multiple Sites	X	X				X	X	
Test Automation							X	
Specification Techniques							X	
Quality Assurance							X	
Management Support							X	
Hardware Quality							X	
Hardware Availability							X	
Tool Quality							X	
Tool Availability						X		
Technical Knowledge							X	
Project Distribution							X	
Interoperability							X	
Security							X	
Reliability							X	
Maths Volume					X			
Language Experience					X			
Environment Experience					X			
Database Size					X			
Programming Techniques					X			
Analyst Capabilities					X			
Applications Experience					X		X	
Reused Code					X			
Language					X		X	
Requirement Volatility					X			
Architecture					X			
Real-Time					X			
Facilitate Change		X					X	
Software Interface		X			X			
Third Party Access		X						
Documentation		X						
User Training		X						

7. Other External Measures

External measures (other than variants of Function Points) have been proposed based on the information available during System Design and Work Allocations. Two of the most promising of these are discussed in separate sections: Application Features is discussed in Section 7.1 and, Work is described in Section 7.2. Several other external measures are discussed below.

Modules, Programs and Sub-systems

Some of the external measures, such as Modules [Anderson, K. J. et al., 1988], Programs [Lederer, A. L. and Prasad, J., 1993a] and Sub-systems [Hufton, D. R., 1985], or combinations of these measures [Lee, H., 1993] are closely related to code-based measures.

They provide a coarser measure of the Size of the system than Lines of Code. The values of the measures depend on the design of the system, but are less influenced by the final implementation of the system and differences between programmers and development technologies, including programming languages, than Lines of Code.

These measures tend to be useful for tracking progress of the system and for allowing estimates to be made of the level of reuse in the system but there is little, if any, empirical evidence to support their use for Effort Estimation.

Files

The number of files, and the number of reports were suggested as measures of Software Size in [Putnam, L. H., 1978], as this worked well with Putnam's Cost estimation model. However, other work [Jeffery, D. R., 1987; Kingston, G. et al., 1995] has shown that Putnam's Cost model may not be appropriate. Therefore we should not assume that Files are a suitable measure of Software Size based on Putnam's work. However, files may still be a suitable measure of Software Size. Some analysis on Function Points (of which Files are a component) has shown that the correlation of Files with Effort is statistically significant, but relatively poor [Kitchenham, B. and Kansala, 1993].

Extensions

Other external measures have been suggested as extensions to existing measures, such as Function Points or Lines of Code. For example, the degree of integration with existing systems may be considered [Lederer, A. L. and Prasad, J., 1993a; Lederer, A. L. and Prasad, J., 1993b]. The properties of most such extensions have not been discussed in the literature in any detail.

Design Models

Other external measures are based on the models of the system available during the design phase. They may either be used directly, or to estimate other Size measures, such as Lines of Code. One of the advantages of these types of measures is that they

are available early, and they may be supported by CASE tools which could be used to automatically calculate the value of the measure [Tate, G. and Verner, J. M., 1991]. They may be based on a single model of the system, or on multiple models including:

- Entity Relationship Diagrams
- Data Flow Diagrams
- User Interface Components.

One problem with these measures is that the models that they are based on may be developed to varying levels of detail, so that the level of detail needs to be specified for the models to be used consistently.

The Size of a software system can also be determined indirectly. For example work-breakdown structures, which identify the tasks to be performed in a software development, can be constructed. They can be used to estimate the Size of the development activity (as opposed to the Size of the Software) and this Size can in turn be used to measure the Effort required for the development [Abdel-Hamid, T. K. et al., 1993]. The drawbacks of this approach are:

- it is subjective,
- there are few empirical studies relating Effort to work-breakdown structures
- it cannot be easily determined for past projects
- it is highly dependent on the approach used, as the tasks will be different if different approaches are taken
- a method of sizing the leaf tasks is required, because the tasks at each level (and the leaf tasks) are not the same the same size
- it is difficult to determine if it is complete, and missing components are difficult to identify.

The advantages are:

- it is constructive,
- the Size is directly attributable to the tasks to be undertaken, so it can be used for tracking progress,
- the Size of new, enhanced and maintenance projects can be determined in the same fashion
- activities which reduce the Size (in Lines of Code) of the Software are easily captured as these activities should appear in the Work Breakdown.

In summary, care needs to be taken when using these measures independently, but they may be useful in conjunction with other measures of Software Size.

7.1 Application Features [Mukhopadhyay, T. and Kekre, S., 1992]

Application Features is a measure of functionality which was developed for Process Control systems.

In some ways, Application Features are similar to Function Points. Both Application Features and Function Points were developed from the perspective of the user. However, as Application Features were developed for Process Control systems and Function Points were developed for Transaction-based systems, they are not variations of each other. Therefore, they are considered separately.

Application Features are a weighted combination of Communication Features, Position Features and Motion Features.

Mukhopadhyay and Kekre provide methods for estimating Unadjusted Function Points and Lines of Code from the components of Application Features and they use these Function Points, or Lines of Code, values to estimate Effort using standard Effort Estimation models, such as COCOMO. They compared these Effort Estimation models with their own models which determined Effort directly from Application Features. They determined that their models were more accurate, and more precise.

Application Features appear to be easier to determine, and less subjective than Function Points.

The authors believe that it is likely that there is less redundancy with Application Features than with Function Points.

As with Function Points, the scope of Application Features is limited, and the authors believe that it is unlikely that Application Features will be decomposable at the module level.

The authors believe that may of the problems with Function Points will also apply to Application Features. However, additional studies would be required to confirm or refute this statement.

7.2 Work [Shepperd, M., 1992]

Shepperd's Work was developed as a measure of Software Size which would be used in the evaluation of designs. The Work measure was designed to capture the Functionality of the software at the module level, and to capture the communication required between the modules to achieve the Functionality.

The Work of a module is then defined as a weighted sum of the number of primitive requirements (P) the module meets and the scheduling overhead (See Equation 5).

The scheduling overhead is determined from the number of primitive (I), and higher level (H), requirements met by the module and its subordinates.

$$\text{Work} = |P| + \alpha(|I| - 1 + |H| - 1)$$

Equation 5: Shepperd's Work Equation for Modules

The Work of the designed system is the sum of the work of its modules. (Note that this is not the same as the Work of the system, considered as a single entity).

Shepperd claims that the Work of the system correlates reasonably well with Lines of Code, but does not consider how well work correlates with Effort.

7.2.1 Measurement

7.2.1.1 Repeatability

Clarity, Objectivity

The definition of what constitutes a requirement could be ambiguous, but once defined, the required information should be able to be determined objectively.

Known Information

The information required should be known after the design is complete. The information would not be known prior to the completion of the design. It also requires a detailed requirements document of the software system to be developed.

Precision, Rater Independence

No studies on the precision of Work calculations have been undertaken. However, it is likely that if an appropriate definition of requirements is determined, then there will be little or no variation between the Work values obtained by different raters, or by different counts performed by the same rater. (Note that as Work is a measure of the Design, both the Requirements Document and the Design must be completed before the Work can be calculated, so they must be independent of the rater.) Compare this with Function Points, where subjective complexity ratings of different modules are required.

Stability

It appears likely that small changes in the requirement, or the design, would result in small changes to the system's Work. However, under certain circumstances (eg when P and α are both relatively small, and H and I are relatively large), small changes in the value of α may result in significant changes to the Work.

7.2.1.2 Soundness

Constructiveness

Work is a constructive measure as differences in the Sizes of different system designs can be traced to differences in the requirements, or differences in the decomposition of the system.

Additivity

The Work measure is additive and inverse additive, because the Work of a system design is the exactly the sum of the Work done by its components.

Minimality

It is not clear if the sub-measures of work - particularly I and H - are independent, and therefore Minimal. It would appear likely that I and H may be highly correlated, and that an alternative sub-measure would be more appropriate.

Consistency

Work was developed for the evaluation of different designs of the same system. It can only be used in the comparison of designs, and it is not clear if designs for different systems should be compared using the Work measure. This suggests that Work may be of limited use for Effort estimation. However, Shepperd showed that Work correlated well with Lines of Code, which are often used for Effort Estimation. Therefore, it appears that Work could be used for Effort Estimation.

7.2.1.3 Other Measurement Properties

Approach

Empirical

Calibration

The coefficient, α, should be calibrated. Shepperd suggests that α depends on the type of application and the development environment, but recommends an initial value of 0.33 based on his experience.

Refinable

The Work measure cannot be refined, although it could be determined for the Source Code as well as the design.

Scale

The Work measure is a function of the requirements in different modules which can be determined absolutely. As Work is a linear combination of the requirements in each module, it is a ratio scale.

Granularity

The granularity of Work is coarser than that of Lines of Code and probably slightly finer than that of Function Points.

Decomposable

As Work is a bottom-up measure, that is the value of the measure for the system is calculated from the values of its components, it is decomposable.

Appropriateness

Work is probably not an appropriate measure of Software Size for Effort estimation early in a Software development, because it cannot be determined before the system is designed.

7.2.2 Use

7.2.2.1 Scope

Shepperd's Work appears to be appropriate for new projects from any domain.

Work requires a methodology where the requirements are known by the end of the design phase, and would be less appropriate for systems whose requirements evolve during the development. The aspects of software which Work covers depend on those covered by the requirements.

As Work is decomposable, it should be possible to use it to measure the process of the development.

It is not clear how Work would be used during the system maintenance or enhancement. The Work for the changed system could be determined, or it may be necessary to also compare the requirements of the original system with the changed system.

7.2.2.2 Calculation

Ease of Measuring or Calculating for New Projects

It should be relatively easy to determine the Work for new projects, once the design phase is complete.

Ease of Obtaining for Past Projects

It should be possible to obtain the Work for past projects where the original requirements, and either the Source Code, or the Design are known. It would be harder to determine the work if requirements have not been traced to the Design or Code.

Software Support

The authors know of no software support for Work. However, it should be possible to automatically determine the Work from a design if a standardised method is used to trace the requirements to the design.

First Phase Applicable

At the completion of the Design.

Database Required

It is preferable that the Work measure is calibrated using a database of past projects.

7.2.2.3 Software Costing

The authors know of no research relating Shepperd's Work to Cost or Effort. Shepperd's Work can be used to estimate Lines Of Code, which could then be used to estimate Effort. However, the accuracy and precision of this approach would be very limited.

8. Combination Measures

The measures discussed in the previous sections have their strengths and limitations. Some researchers have tried to combine internal and external measures to try and capture the benefits of both types of measures. Several methods of combining measures have been attempted with varying levels of success. These include: combining a group of measures into a single measure, with a single value; comparing measures obtained by different raters and approaches; using vectors of Size measures. The most sophisticated method proposed is discussed in Section 8.1. It uses different vectors of Size measures at different phases in the development.

Single Valued Combinations

One combination measure which results in a single value was proposed in [Putnam, L. H., 1978]: a combination of sub-programs and, reports or files. It is not clear from the literature what benefits, if any, this measure offers over other approaches.

This measure is similar to the Modules measure discussed in Section 7, because it should be possible to estimate sub-programs (an internal measure) reasonably accurately from the detailed design of the system. The properties are therefore similar. It is a coarse, design and technology dependent, measure of Software Size, on which few studies of Effort Estimation have been performed.

Multiple Raters

The Software Productivity Consortium, [SPC, 1994], recommends using different approaches and people to estimate the Size of the Software in Lines of Code. If the estimates are similar, then they are used. If they are not, then the cause of the difference should to be identified, or the estimates should be weighted according to the likelihood of their correctness. They advocate choosing the weighting factors based on expert opinion, but the authors suggest that it should also be possible to use statistical techniques to determine the appropriate weights.

Size Vectors

The use of vectors of different measures of Size is discussed in [Conte, S. D. et al., 1986]. As vectors induce a partial ordering on the system, they do not recommend their use. There are, however, advantages to using a vector of Size measures for Effort estimation. Keeping a Size vector, rather than combining the measures, enables the Effort model to be calibrated in a single step. If the elements of the vector were first combined into a single Size measure, then this would need to be calibrated before the local parameters for the Effort model could be determined.

Kulkarni et al (Section 8.1) take Size vectors one step further, using different vectors at different stages in the development.

8.1 Phase Driven Size Estimation [Kulkarni, A. et al., 1988]

Rather than rely on a single measure of Size, this method uses several measures of Size appropriate for each phase in the development process. The Size for each phase is a vector containing the number of each type of output for the phase. It is assumed that the number of outputs can be determined from the inputs to the phase, according to some average values which are calculated using matrix manipulations. These matrices can be combined to determine the Size (number) of outputs of successive phases in the development.

The Software Productivity Consortium, [SPC, 1994] also advocates using different Size measures at different stages of the software development. Their approach differs slightly in that the inputs and outputs of each phase are not measured. Instead different measures are used at milestones in the project. These measures need not be correlated in the manner advocated by Kulkarni et al.

8.1.1 Basis

Purpose

This approach to Software Size measurement allows Size to be tracked between the different phases in the development.

Class

Varied

Theory

This approach relies on Software Development being similar to the development on a production line, where the number of outputs from each phase can be determined from the number of inputs. While the authors believe that, in general, this is not a valid assumption for software development; it is not clear without some experimentation, if it is feasible in these circumstances.

Source

External in the early stages and internal in the later stages.

8.1.2 Measurement

8.1.2.1 Repeatability

Clarity

The approach is only loosely defined. The measures of Size to be used for each phase, and their correlations, are not specified. These need to be determined for each

development process and therefore they do not need to be more precisely defined in the general context.

Objectivity

The process of transforming one Size measure to another is objective. The calculation of the Size measures may or may not be objective, depending on the measures used in a particular instantiation of the approach.

Known Information

The Size measures for each phase should be chosen so that they use only information known at the completion of the phase. The Size of the inputs to each phase should be based on known information.

Precision, Rater Independence and Stability

These depend on the measures of Size used for each phase. However, no method was suggested for incorporating measures of the precision, or uncertainty, in the Size measures estimated from, or used as input to, the model.

8.1.2.2 Soundness

The Soundness properties depend on the measures of Size used for each phase.

8.1.2.3 Other Measurement Properties

Approach

The approach to transforming the Size measures is empirical. It is suggested that the measures for each phase are direct, or simple empirical measures.

Calibration

The approach requires calibration to determine the matrices to transform from one Size measure to another.

Refinable

The approach allows initial Size estimates of the end product(s) to be determined from the information known at the start of the project by applying a sequence of matrices to an initial Size vector.

These estimates can be refined at the end of each phase using the actual Size of the outputs to the phase, rather than the estimates obtained using matrix manipulation.

The approach does not consider the precision of the estimates, so the precision cannot be refined.

Scale

As the approach is treated as a ratio scale, the Size measures should either be direct measures, or measures which have a ratio scale. However, the scale of the approach depends on the scales of the Size measures used for each phase.

Granularity, Decomposable and Appropriateness

These properties depend on the Size measures used.

8.1.3 Use

8.1.3.1 Scope

The scope of the approach depends on the scopes of the Size measures used for each phase.

8.1.3.2 Calculation

Ease of Calculation

The transformation of the different Size measures can be easily performed for new and past projects. It could be automated.

The measures used determine how easily the Size of each phase can be calculated.

First Phase Applicable

The approach may be used at all phases of the development.

Database Required

A database of past projects is required to calibrate the approach for each development process used.

8.1.3.3 Software Costing

Kulkarni et al suggest that a single Size measure for each phase is chosen as the primary Effort Estimator. Estimation models could then be developed for each of these Estimators.

9. Growth Measures

The estimated Size of a system typically changes during its development. This may be due to changes in the system's requirements or it may just be due to increased knowledge about the system. Growth measures are designed to capture how the Size of the system changes during its development. Little work has been done on growth measures. The usual approach is to consider the difference between the initial estimate and the Size of the final product (see Equation 6) [SPC, 1994]. This factor may be applied to future estimates to try and correct them.

This type of growth factor should have an average value of 1 for a properly calibrated empirical Sizing approach. However, it has been stated that the duration of a software project is influenced by the initial estimate [Abdel-Hamid, T. K., 1993], and this may also be true for Size estimates.

$$Growth = \frac{Delivered\_Size - Initial\_Estimate}{Initial\_Estimate}$$

Equation 6: A Standard Growth Equation

A more useful Growth model would identify systems which were growing and whether the rate of growth was accelerating or decelerating. If the rate of growth accelerates during the later phases of the development then the risk of an unsuccessful project increases. The authors know of no such model of Software growth.

10. Discussion

The different Sizing techniques presented in this paper have different properties as captured by the criteria introduced in Section 3. However, it is not always clear which properties are most appropriate for measures of Software Size. Furthermore, some of the properties are highly correlated, so it may not be possible to obtain a measure with all the desirable properties.

This section first discusses the strengths and limitations of the properties (which relate to the criteria given in Section 3) in general, and in their application to Software Costing. The properties of the existing Size measures are then summarised and desirable and undesirable properties are noted. The properties are further analysed to try and determine if it possible to have a measure with all the desirable properties. This information is then used to determine which features of existing models should be incorporated into a new model of Software Size specifically for Effort Estimation.

10.1 Desirability of Properties

The section reviews the criteria introduced in Section 3 to determine which values are desirable in measures of Software Size for Effort Estimation.

The properties of the existing Size measures are summarised according to their desirability in Section 10.2.

10.1.1 Basis

10.1.1.1 Purpose

Measures which were developed specifically for Effort Estimation are the most desirable. However, none of the measures investigated were developed for this purpose. Therefore a theory to relate the original purpose or an attempt to valid the measures for Effort Estimation should be provided. (See Theory.)

10.1.1.2 Class

Section 10.2 investigates the properties which are commonly related to the different Classes of Software Size measure. The Class of the measure is not important in any other manner.

10.1.1.3 Theory

While some of the measures were based on theory or stated assumptions which were valid for their initial purpose (see Purpose), there was no theoretic basis for the use of any of the measures for Effort Estimation.

A theoretical basis aids the understanding of when the measure can be used and its strengths and limitations. It is therefore desirable for a measure to have a theoretical basis which relates to the purpose for which it is being used.

10.1.1.4 Source

The source of the measure is of little importance for Effort estimation provided there is a theory to relate Effort to the source (see also Theory), and that information about the source is available at the appropriate times in the development cycle (see also Known Information and First Phase Applicable). Bearing these considerations in mind, measures which are not based on the Source Code (which excludes the internal measures) are preferred.

10.1.2 Measurement

10.1.2.1 Repeatability

Clarity

Clarity is a desirable property for Software Size measures.

One consequence of a poorly defined measure of Software Size is that the definition can be interpreted in a variety of manners. This may lead to a family of Software Size measures being created from the original definition. These measures are often referred to by their family name even though each measure may be customised for the local environment. Extra care needs to be taken when developing - and using - models where the Software Size measure belongs to such a family. This is because statistical techniques assume that the data they are manipulating belongs to a population.

Note that Clarity in the definition of a measure is not the same as Rater Independence of that measure. A definition may be clear, but the measure may still be Rater Dependent. For example, the definition of IFPUG Function Points [Morris, P., 1994] is relatively clear. However, it can be seen that the measure is still dependent on the rater when: the Rater is required to derive their own functional decomposition; and to assign (subjectively) the complexity of the functions.

A clear definition is usually necessary for a measure to be Rater Independent.

Objectivity

It is difficult to define measures which are based on subjective assessments so that the assessments are likely to be similar, or preferably identical. For example, it is difficult to assess the complexity of a module before it is designed or coded.

The values given for subjective measures such as this vary between raters, and can even vary for a given rater (depending on their mood).

Subjective factors which have a large effect on the final value of a measure should be avoided wherever possible. If such factors are not avoided it is difficult to trust the value obtained for the measure unless the uncertainty in its value is captured.

Known Information

Known information is a desirable property for measures of Software Size. If estimated - rather than actual - values of sub-measures are used, then there is a high degree of uncertainty in the value of the final measure. While in some circumstances, it may be possible to quantify this uncertainty, it is likely that the uncertainty in any Effort estimate based on such a measure of Size would be unacceptably large.

Rater Independence

If the values of measures obtained by different raters (measurers) are to be compared, then it is desirable for the measure to be Rater Independent.

Even within an organisation, the use of a single rater is not a suitable substitute for the rater independence of a measure. If the rater leaves, then the historical data would need to be recalibrated, or discarded.

If all raters in an organisation regularly check each other's measurements, then it is likely that they will tend to use a rater independent measure. New raters could then be trained using a master and apprentice scheme. Of course, extra care is required if measures are to be compared between organisations, or even across a single organisation.

Stability

It is important for a measure of Software Size to be stable, that is have small variations for small variations in its sub-measures. This is particularly important if the sub-measures are subjective, estimated, or rater-dependent. However, errors can be made in the values of all sub-measures.

For example, consider (raw) Function Points. The number of Function Points of a system can be automatically determined from an Annotated Functional Decomposition (AFD) of the system. However, it is possible to derive multiple, correct, AFDs for the same system (as well as incorrect and incomplete AFDs). If two different, but correct, AFDs exist for a system, it is desirable that the associated Function Point values be the same, or at least similar. Therefore, the Function Point measure should be stable.

10.1.2.2 Soundness

Constructiveness

Three of the properties: Constructiveness, Decomposable and Additivity are similar, but they take different values and may not all be desirable in a Sizing approach for Effort estimation.

A measure is Constructive if differences in its values can be traced to differences in the values of sub-measures.

This property is desirable in a measure of Size for Effort Estimation as it allows differences in the scope of the system (or in the sub-measures) to be compared and forms a basis for determining which system is the most appropriate or cost effective to build.

Additivity

A measure is Additive if the value for a combination of components or sub-systems is at least the sum of the values of the components.

While [Symons, C. R., 1988] recommends that Size measures should be additive, the authors believe that the property requires further investigation.

Consider a system which may be implemented as either one application (A) or two applications (B and C). If there is no difference in the real functionality between the two implementations then A can be implemented using B and C with a top level routine which allows users to choose between B and C.

The authors believe that the Effort to produce the single system A should be, at most, the Effort to produce the two applications B and C separately, as optimisations may be possible in the single system. The functionality of the combined system A is probably less than the sum of the functionality of the two applications B and C as there may be some overlap between the two applications. Finally, the length of A will depend on which overlaps between B and C, and optimisations, are identified and implemented.

Therefore, Size (or at least Size measures of Class Functionality and Length) should not be additive and the Size of a system with two components or sub-systems should be at most the sum of the Sizes of the components (inverse additive). (Note however that Size measures when the sum of the Sizes of two components is equal to the Size of the combination of the two components are both additive and inverse additive. These measures may be appropriate for Software Sizing.)

Minimality

The sub-measures of a measure of Software Size should be Minimal, that is none of the sub-measures should be redundant. If some of the sub-measures are redundant then it is impossible to uniquely determine the influence of the sub-measures on the Size measure. Furthermore, if a minimal set of sub-measures are highly correlated, then it is difficult, if not impossible to determine the influence of the correlated sub-measures.

Consistency

If the values for a measure of Software Size are being used to compare two or more systems, then the measure must be consistent across the domains and environments of the systems. That is, the value obtained for the Size of System A should only be bigger than the value for System B if System A is "bigger", in some appropriate sense, than System B.

For example, it can be argued that if System A is written in C and System B is written in C++ then Lines of Code should not be used to compare the Sizes of the systems.

Completeness

If a measure of Software Size is to be used in different environments and domains, it is also important that it is complete. It is, however, difficult to determine if a measure of Size is complete without an underlying theory.

Furthermore, when using Sizing for Effort estimation, it is often not clear whether factors should be incorporated in the Size of the Software, or should be separate factors in the Effort estimation models.

This is one reason why it is not clear whether such Size measures should be simple, or allow for variations in many factors. (There are also other reasons, similar to those given for Granularity.) For example, the complexity of Function Point Adjustment Factors has been debated. Researchers have argued for more complex [Symons, C. R., 1988] and less complex (as used in SPQR) [Jeffery, D. R. et al., 1993] Function Point Adjustment Factors.

Approach

Where possible, Software Size should be measured directly. However, when this is not possible, the approach should be based on statistical, or theoretical foundations.

Most of the other approaches to measuring Software Size considered in this paper are based on algorithms derived from expert opinion. While this approach may be valid within an organisation, it is not clear if such expertise can be transferred to different development environments or to different types of application.

10.1.2.3 Other

Refinable

If a Size measure is refinable, (variants of) the same measure can be used early in the software development, and later in the life-cycle. This enables changes in the scope of the project to be tracked, as well as early estimates of the Size of the project.

Most of the approaches examined in this paper provide a measure of Software Size at a single point in time. Correlations with other measures (such as Lines of Code) may be given. These tend to be based on averages.

Without a theoretical basis for comparison it is difficult, if not impossible, to determine if deviations from the average are random, or due to changes in the scope of the software.

Calibration

It has been shown that local data and calibration are required for Effort Estimation [Jeffery, D. R. and Low, G., 1990]. It has also been suggested that calibration is required for some models of Software Size (such as Function Points) [Symons, C. R., 1988].

However, Software Size models are difficult to calibrate independently of the calibration of Effort Estimation models. To calibrate both simultaneously requires large amounts of data. Thus, the authors prefer Software Size models which are not amenable to calibration because they either a) are a direct measure or b) have a theoretical foundation.

The authors acknowledge that calibration may be required for other measures of Software Size, particularly those which can be refined during the Software Development.

Scale

The scale of a measure of Software Size determines the manner in which its values can be manipulated. Ratio scale measures are generally preferred as these are required for the application of standard statistical techniques, such as linear regression. However, the scale of a measure is less important than the appropriate treatment of the measure for its scale.

Granularity

A more complicated question which needs to be considered is: should the granularity of Software Size measures be coarse or fine?

The best measures are those which can be used at both coarse and fine levels of granularity. For example, consider distance. The distance a car has travelled is normally measure in kilometres, but the distance between atoms in a molecule is measured in nanometres.

While it is desirable that a measure of Software Size is capable of being measured with a fine level of granularity, there are problems when the measure is developed to too fine a level of granularity too soon.

Measures are normally developed in several stages. In the first stage, the measures are coarse, and often ordinal. For example, people were classified as tall or short before their height was measured. As the concepts and theory underpinning the measure developed, the granularity of the measure became finer.

We currently have a poor understanding of the essence of Software Size. It is not clear which artefacts should be measured, nor is it clear what aspects of Size are important for Effort Estimation. Therefore, we should proceed slowly, and first

develop coarse measures for Software Size. As we develop a theory of Software Size, the measures will naturally become finer.

The use of a coarse measure can also be justified by practical considerations. Only imprecise Effort estimates can be made based on the Software Sizing approaches considered in this paper. However, by precisely specifying the Size of the Software there is a natural tendency to over-specify the precision of the Effort Estimate obtained.

This is particularly important when estimates are made early in the development. As Software is often considered to be a flexible commodity, changes to the requirements are often made in the middle of the development, and the initial requirements are often poorly understood. Thus, the Size of the Software is likely to change from the initial estimate.

It appears inappropriate to spend resources on determining a precise (fine granularity) measure of Size, which will change with the requirements. A coarse measure, which would not change with small variations in the requirements would appear more appropriate as an early measure of Software Size. Later in the development, when the requirements are more stable, it may be appropriate to refine the measure to a finer level of granularity.

Decomposable

A measure is Decomposable if the value for a system can be attributed to the values for the components. This property is desirable, particularly for keeping track of progress.

Appropriateness

None of the Software Size measures evaluated in this paper were derived, modified, or validated as a measure for Effort Estimation. Therefore the appropriateness of all the measures is limited.

However, appropriateness is an important measurement theoretic consideration which should be addressed for future measures of Software Size.

10.1.3 Use

10.1.3.1 Scope

The scopes of most of the Software Size measures considered in this paper are poorly defined.

To ensure that the values of the measure accurately reflect the Size of the system, the scope of Software Size measures should be more precisely defined.

Software Size measures with small scope have the potential to be more accurate and precise, but their domain of applicability is limited. Software Size measures with larger scope offer the opportunity to compare the Size of more systems, but often lack accuracy and precision.

However, for an organisation like the Department of Defence, which needs to remain at the cutting edge of technology, and which sub-contracts out the development of most its Software Systems Development, the authors believe that a standard measure of Software Size, whose Scope is not limited (eg by application domain, development environment, language or development team) is desirable.

10.1.3.2 Calculation

Ease of Deriving

In order for a Software Size measure to be used (in practice) for Effort Estimation, the measure should be easy to derive. This applies to both new projects for which Effort Estimates are to be made, and for past projects if the Estimation approach is to be calibrated.

First Phase Applicable

However, it is generally more important that the Effort Estimations can be made early in the development cycle, rather than the Size measure being easy to obtain for past projects.

Database Required

If a database is required (eg to calibrate a measure of Software Size) then the measure cannot be used before data on previous projects has been collected.

Size measures which do not solely rely on data from previous projects are preferred by the authors. (For more details, see the notes under Calibration in Section 10.1.2.3).

Software Support

Software Support is one method which can be used to reduce the time, effort and complexity of determining the value for a measure of Software Size. However, it is not significant in its own. (For example, tools intended to support the task, but which actually made the task more difficult, would tend to be ignored.)

10.1.3.3 Effort Estimation

Correlation with Effort

This paper focused on measures of Software Size for use in Effort Estimation.

In this context, Correlation with Effort is an important property of the measure. It is not anticipated that any measure of Software Size would show 100% correlation with Effort. However, measures with higher correlations would be preferred to other measures available at the same phase in the life-cycle with lower correlations.

Unfortunately, there was limited information on the Correlation with Effort of many of the measures of Software Size considered in this paper.

Precision of Estimates

The precision of Effort Estimates is expected to vary depending on three related factors: 1) the phase in the development when the estimate was made, 2) the

available information on which the estimate could be based and, 3) the time spent making the estimate. More precise estimates can be expected from measures of Size which are available later in the Software Development.

The precision of Software Size measures is important as measures which are imprecise are associated with high degrees of risk.

Accuracy of Estimates

The accuracy of Effort Estimation models should not depend on the measure of Software Size on which they are based, but rather on the method (statistical or otherwise) used to develop the model.

As accurate models are desirable (so that sound, risk-managed decisions can be based on the results of their application), care should be taken in the process used to develop Effort Estimation models regardless of the measure of Software Size used.

The precision of an Effort Estimation model, however, is likely to depend on the measure of Software Size on which it is based. Note that imprecise models may appear to be inaccurate when the estimated and actual Effort are compared for a small set of projects.

10.2 Properties of Different Classes of Size Measures

The properties of the different Size measures tend to be associated with their Class, as shown in Table 3. Some of the properties, such as the measures' limited theoretical foundations are consistent across all of the classes. These are shown in italics. Properties which are considered desirable are indicated by a '+' and those which are considered undesirable are indicated by a '-'.

Property Trade-offs and Patterns

Desirable properties which tend to be grouped together for the Software Size Measures considered in this paper are given in Table 2.

The properties in the left column of Table 2 tend to be associated with Functionality measures, and those in the right with Length measures.

It is possible that no one Size measure will meet all these criteria, but there is no obvious reason for the two sets of criteria to be mutually exclusive. For example, it should be possible for a measure of Software Size to be both Constructive and Decomposable.

Indeed, the Sizing approach of [Kulkarni, A. et al., 1988], where Size measures are tracked for different artefacts at the various stages in the development, has properties which span these sets. This approach differs from the other Length, Complexity and Functionality approaches presented in that 1) it was developed to allow Effort and Size estimates to be refined during the Software Development, and that 2) it considers all artefacts, not just the Source Code. The authors believe that if

Software Sizing methods are to improve, then one or both of these approaches need to be investigated further.

Table 4: Property Groupings

Constructive	Clear
Applicable early in the development	Automatable
Consistent	Decomposable
	Can measure progress
	Minimal
	No calibration
	Any domain
	Easy for past projects

For the moment, however, measures of Software Size should be chosen according to the purpose for, and the time at which, the measure is to be used.

10.3 Future Directions

The authors believe that future work should investigate measures for Software Size which are applicable early in Software Development, and which can be refined as the development progresses. These measures should also be measures of Software Scope. That is, they should be independent of the development environment and implementation. The current Functionality measures are the closest - of the Software Size measures - to measures of Software Scope. Thus, it is recommended that future measures should arise through consideration of Functionality rather than Length or Complexity.

The Australian Department of Defence would benefit from such advances. They will make it easier to track the progress of projects from their original requirements and will enable Cost Estimates to be refined - and compared - throughout the Software Development.

A new concept of Software Size, called Capacity is being developed as part of the iMAPS task [Burke, M. M., 1993]. Capacity will be capable of being measured in a variety of ways at different stages in the Software Development. It is hoped that Capacity will overcome some of the limitations of previous measures of Software Size and enable more precise Effort estimates to be made.

Table 5: Common properties of the different classes of Size measures

CLASS:	Length	Complexity	Functionality
BASIS			
Purpose	• Storage	• Maintenance or Testing	• User Understanding
Source	• Internal	• Internal	• External
Theory	• *Limited*	• *Limited*	• *Limited*
MEASUREMENT			
Repeatability	+ Many clear, unambiguous definitions + Automated • Relies on the source code + *Stable*	+ Unambiguous definition(s) • Automation varied • May rely on code, or decompositions + *Stable, but small range of values*	• Single ambiguous definition – Limited automation • Relies on functional decomposition + *Stable*
Soundness	– Not constructive + Additive and inverse-additive + Minimal – Not consistent across languages and definitions	– Not constructive • Additivity varies + Minimal • Consistency varied	+ Constructive + Inverse additive – Redundancy + Consistent, but depends on how calibrated
Approach	+ Direct	• Algorithmic	• Algorithmic
Other	+ Estimates may be refined to actuals + No Calibration • Finest granularity + Decomposable + Ratio (direct) scale	• No refinement possible + No Calibration • Coarsest granularity • Varied + Ratio scale	+ Limited refinements possible – Calibration required • Coarse granularity – Not decomposable – Scale invalid, but treated as a ratio scale.
USE			
Scope	+ Any domain + *Any language* • New projects or enhancements + Can be used to measure progress • Source code	+ Any domain + *Any language* • New projects or maintenance – Cannot be used to measure progress • Source code	– Limited domains + *Any language* • New projects or enhancements – Difficult to use to measure progress • Source code
Calculation	+ Easy to derive for current and past projects	• Varied ease to derive for current and past projects	• Easy to derive for new and some past projects, but may take some time
Effort Estimation	– Imprecise	• Limited information available	– Imprecise

11. Acknowledgements

The authors thank Stephen Fry for his help with the literature search and, Stefan Landherr, Rudi Vernik and Peter Fisher for their contributions in related discussions.

12. References

[Abdel-Hamid, T. K., 1993] "Adapting, Correcting and Perfecting Software Estimates, A Maintenance Metaphor." *IEEE Computer* Mar: pp20-29.

[Abdel-Hamid, T. K. et al., 1993] "Software Project Control: An Experimental Investigation of Judgment with Fallible Information." *IEEE Transactions on Software Engineering* 19(6): pp603-612.

[Abran, A. and Robillard, P. N., 1993] "Reliability of Function Points Productivity Model for Enhancement Projects (A Field Study)". *Conference on Software Maintenance*, Montreal, Quebec, Canada, IEEE Computer Society Press.

[Abran, A. and Robillard, P. N., 1994] "Function Points: A Study of Their Measurement Processes and Scale Transformations." *Journal of Systems and Software* 25(2): pp171-184.

[Albrecht, A. J. and Gaffney, J. E., 1983] "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE Transactions on Software Engineering* SE-9(6): pp639-648.

[Anderson, K. J. et al., 1988] "Reuse of Software Modules." *AT&T Technical Journal* Jul/Aug: pp71-76.

[Arifoglu, A., 1993] "A Methodology for Software Cost Estimation." *ACM SIGSOFT Software Engineering Notes* 18(2): pp96-105.

[ASMA, 1994] ASMA Project Database, ASMA.

[Banker, R. D. et al., 1994] "Automating Output Size and Reuse Metrics in a Repository-Based Computer-Aided Software Engineering (CASE) Environment." *IEEE Transactions on Software Engineering* 20(3): pp169-187.

[Betteridge, R., 1992] "Successful Experience of using Function Points to Estimate Project Costs Early in the Life-Cycle." *Information and Software Technology* 34(10): pp655-658.

[Boehm, B. W., 1984] "Software Engineering Economics." *IEEE Transactions on Software Engineering* SE-10(1): pp4-21.

[Boehm, B. W., 1988] "Improving Software Productivity". *Improving Productivity in EDP System Development.*

[Boehm, B. W. and Papaccio, P. N., 1988] "Understanding and Controlling Software Costs." *IEEE Transactions on Software Engineering* 14(10): pp1462-1477.

[Boehm, B. W. and Wolverton, R. W., 1980] "Software Cost Modelling: Some lessons learned." *Journal of Systems and Software* 1: pp195-201.

[Bourque, P. and Cote, V., 1991] "An Experiment in Software Sizing with Structured Analysis Metrics." *Journal of Systems Software* 15: pp159-172.

[Britcher, R. M. and Gaffney, J. E., 1985] "Reliable Size Estimates for Software Systems Decomposed as State Machines". *Nineth International Computer Software and Applications Conference*, Chicago, Illinois, IEEE.

[Brooks, F. P., 1987] "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer* (Apr): pp10-19.

[Burke, M. M., 1993] iMAPS Task Plan (No. DST 93/949), DSTO.

[Camelotti, W., 1995] 'Tis The Season To Be Jolly... & Safe! Health Yourself. **Dec**: pp4-5.

[Charismatek, 1994] Function Point Workbench - For the People Who Count. User Manual., Charismatek Software Metrics.

[Conte, S. D. et al., 1986] Software Engineering Metrics and Models. Menlo Park, Benjamin/Cummings.

[Fairley, R. E., 1992] "Recent Advances in Software Estimation Techniques". *14th International Conference on Software Engineering*, Melbourne, Australia, ACM.

[Fenton, N., 1991] Software Metrics: A Rigorous Approach, Chapman & Hall.

[Ferens, D. V. and Gurner, R. B., 1992] "An Evaluation of Three Function Point Models for Software Estimation". *IEEE 1992 National Aerospace and Electronics Conference.*

[Fitsos, G. P., 1980] "Vocabulary Effects in Software Science". *COMPSAC 1980*, IEEE.

[Freiman, F. R. and Park, R. E., 1979] "The Price Software Cost Model". *IEEE 1979 National Aerospace and Electronics Conference*, Dayton, OH, USA, IEEE.

[Friedman, M. A. et al., 1995] Reliability of Software Intensive Systems. Park Ridge, Noyes Data Corporation.

[Halstead, M. H., 1977] Elements of Software Science, Elsevier North-Holland Inc.

[Hastings, T., 1995] "Adapting Function Points to Contemporary Software Systems: A Review of Proposals". *Australian Conference on Software Metrics*, Sydney, Australia, Australian Software Metrics Association.

[Heemstra, 1990] "Software Cost Estimation Models". *IEEE*, IEEE.

[Heemstra, F. J., 1992] "Software Cost Estimation." *Information and Software Technology* 34(10): pp627-639.

[Hufton, D. R., 1985] A Guide to Preparing Technical Estimates for Proposals, Technical (No. 8266/MPR-3), CAP.

[Hughes, R., 1994] "An Exploration of the Problems of Using Function Points Mark II." *Information Systems Journal* 4(3): pp169-183.

[IEEE Std 610.12, 1990] Standard Glossary of Software Engineering Terminology. IEEE Standards Collection, Software Engineering. New York, IEEE. 1993, ed.

[Itakura, M. and Takayanagi, A., 1982] "A Model for Estimating Program Size and its Evaluation". *6th International Conference on Software Engineering*, Tokyo, Japan, IEEE.

[Jeffery, D. R., 1987] "Time Sensitive Cost Models in the Commercial MIS Environment." *IEEE Transactions on Software Engineering* SE-13(7): pp852-859.

[Jeffery, D. R. and Low, G., 1990] "Calibrating Estimation Tools for Software Development." *Software Engineering Journal* 5(July): pp215-221.

[Jeffery, D. R. et al., 1993] "A Comparison of Function Point Counting Techniques." *IEEE Transactions on Software Engineering* 19(5): pp529-532.

[Jones, C., 1988] A Language Level to Fit the Job. ComputerWorld: pp21-24.

[Jones, C., 1994] "Cutting the High Cost of Software "Paperwork"." : pp79.

[Jones, C., 1995a] Programming Languages Table, SPC.

[Jones, C., 1995b] What are Function Points?, WWW Report, Software Productivity Research, Inc.

[Kaplan, H. T., 1991] "Ada COCOMO Cost Estimating Model and VASTT Development Estimates vs. Actuals." *Vitro Technical Journal* 9(1): pp48-60.

[Kemerer, 1991] Software Cost Estimation Models. Software Engineer's Reference Book, Butterworth-Heinemann Ltd.

[Kemerer, C. F., 1987] "An Emperical Validation of Software Cost Estimation Models." *Communications of the ACM* 30(5): pp416-429.

[Kemerer, C. F., 1993] "Reliability of Function Points Measurement: A Field Experiment." *Communications of the ACM* 36(2): pp85-97.

[Kingston, G. et al., 1995] "On the Statistical Significance of Productivity Factors in Software Development Effort Predicition.". *Australian Conference on Software Metrics*, Sydney, Australia, Australian Software Metrics Association.

[Kitchenham, B. and Kansala, 1993] "Inter-item Correlations Among Function Points". *IEEE Metrics Symposium*, IEEE.

[Kitchenham, B. A., 1992] "Empirical Studies of Assumptions that Underlie Software Cost-estimation Models." *Information and Software Technology* 34(4): pp211-218.

[Kitchenham, B. A. and Taylor, N. R., 1984] "Software Cost Models." *ICL Technical Journal* (May): pp73-102.

[Kitzberger, B., 1995] Re: lines of code (was Re: The SEI's CMM-Flawed).

[Kok, P. A. M. et al., 1990] "The MERMAID Approach to Software Cost Estimation". *ESPRIT Technical Week.*

[Kulkarni, A. et al., 1988] "A Generic Technique for Developing a Software Sizing and Effort Estimation Model". *IEEE*, IEEE.

[Kusters, R. J. et al., 1990] "Are Software Cost Estimation Models Accurate?" *Information and Software Technology* 32(3): pp187-190.

[Laranjeira, L. A., 1990] "Software Size Estimation of Object-Oriented Systems." *IEEE Transactions on Software Engineering* 16(5): pp510-522.

[Lederer, A. L. and Prasad, J., 1993a] "Information Systems Software Cost Estimating: A Current Assessment." *Journal of Information Technology* 8: pp22-33.

[Lederer, A. L. and Prasad, J., 1993b] "Systems Development and Cost Estimating Challenges and Guidelines." *Information Systems Management* (Fall): pp37-41.

[Lee, H., 1993] "A Structured Methodology for Software Development Effort Prediction Using the Analytical Hierarchy Process." *Journal of Systems Software* 21: pp179-186.

[Lesnoski, T. M., 1992] "Life Cycle Cost (LCC) Estimating for Large Management Information Systems". *IEEE 1992 National Aerospace and Electronics Conference*, Dayton, OH, USA, IEEE.

[Levitin, A. V., 1987] "Investigating Predictability of Program Size". *Eleventh Annual International Computer Software and Applications Conference*, Tokyo, Japan, IEEE.

[Lo, R. et al., 1995] "Sizing and Estimating the Coding and Unit Testing Effort for GUI Systems". *Australian Conference on Software Metrics*, Sydney, Australia, Australian Software Metrics Association.

[Lokan, C., 1995] "A Critical Examination of Software Size Metrics". *Australian Conference on Software Metrics*, Sydney, Australia, Australian Software Metrics Association.

[Lokan, C. J., 1993] "Software Size Estimation: A Review". *First Australian Conference on Software Metrics (ACOSM'93)*, Sydney, Australia, Australian Software Metrics Association.

[Matson, J. E. et al., 1994] "Software Development Cost Estimation Using Function Points." *IEEE Transactions on Software Engineering* 20(4): pp275-286.

[Matson, J. E. and Mellichamp, J. M., 1993] "An Object-Oriented Tool for Function Point Analysis." *Expert Systems* 10(1): pp3-14.

[McCabe, T., 1976] "A Complexity Measure." *IEEE Transactions on Software Engineering* SE-2(4): pp308-320.

[Morris, P., 1994] Applying Function Point Analysis, Charismatek Software Metrics.

[Mukhopadhyay, T. and Kekre, S., 1992] "Software Effort Models for Early Estimation of Process Control Applications." *IEEE Transactions on Software Engineering* 18(10): pp915-924.

[Nemecek, S. and Bemley, J., 1993] "A Model for Estimating the Cost of AI Software Development: What to do if there are no Lines of Code?". *IEEE International Conference on Developing and Managing Intelligent System Projects*, IEEE.

[Putnam, L. H., 1978] "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." *IEEE Transactions on Software Engineering* SE-4(4): pp345-361.

[Rask, R. et al., 1993] "Simulation and Comparison of Albrecht's Function Point and DeMarco's Function Bang Metrics in a CASE Environment." *IEEE Transactions on Software Engineering* 19(7): pp661-671.

[Reifer, D. J., 1988] Asset-R: A Function Point Sizing Tool for Scientific and Real-Time Systems, Prepared for Journal of Systems and Software (No. RCI TN 299A), Reifer Consultants, Inc.

[Schooneveldt, M. et al., 1995] "Measuring the Size of Object-Oriented Systems". *Australian Conference on Software Metrics*, Sydney, Australia, Australian Software Metrics Association.

[Shepperd, M., 1992] "Measurement of Structure and Size of Software Designs." *Information and Software Technology* 34(11): pp756-762.

[Sneed, H., 1994] Calculating Software Project Costs Using Data Points. Unpublished.

[SPC, 1994] Software Measurement Guidebook, SPC.

[Stathis, J. and Jeffery, D. R., 1993] "An Empirical Study of Albrecht Function Points". *First Australian Conference on Software Metrics*, Sydney, Australia, Australian Software Metrics Association (ASMA).

[Sykes (Ed.), J. B., 1977] Concise Oxford Dictionary. London, Book Club Associates.

[Symons, C. R., 1988] "Function Point Analysis: Difficulties and Improvements." *IEEE Transactions on Software Engineering* 14(1): pp2-11.

[Tate, G. and Verner, J. M., 1991] "Approaches to Measuring Size of Application Product with CASE Tools." *Information and Software Technology* 33(9): pp622-628.

[USC, 1995] COCOMO 2.0 Model User's Manual (No. Version 1.1), University of Southern California.

[Verner, J. and Tate, G., 1988] "Estimating Size and Effort in Fourth-Generation Development." *IEEE Software* (July): pp15-22.

[Verner, J. and Tate, G., 1992] "A Software Size Model." *IEEE Transactions on Software Engineering* 18(4): pp265-278.

[Verner, J. M. et al., 1989] "Technology Dependence in Function Point Analysis: A Case Study and Critical Review". *11th International Conference on Software Engineering*, ACM.

[Whitmire, S. A., 1992] "3-D Function Points". *IFPUG 1992 Spring Conference.*

[Yau, C. and Tsoi, R. H. L., 1994] "Assessing the Fuzziness of General System Characteristics in Estimating Software Size". *Second Australian and New Zealand Conference on Intelligent Information Systems*, Brisbane, Australia, IEEE.

iMAPS: A Review of Software Sizing for Cost Estimation

Gina Kingston and Martin Burke

(DSTO-TR-0360)

DISTRIBUTION LIST

	Number of Copies
AUSTRALIA	
DEFENCE ORGANISATION	
S&T Program	
Chief Defence Scientist)	
FAS Science Policy)	1 shared copy
AS Science Industry External Relations)	
AS Science Corporate Management)	
Counsellor, Defence Science, London	Doc Control sheet
Counsellor, Defence Science, Washington	1
Senior Defence Scientific Adviser)	1 shared copy
Scientific Adviser - Policy and Command)	
Assistant Secretary Scientific and Technical Analysis	1
Navy Scientific Adviser	3 copies of Doc Control sheet and 1 distribution list
Scientific Adviser - Army	Doc Control sheet and 1 distribution list
Air Force Scientific Adviser	1
Director Trials	1
Director, Science Policy - Force Development and Industry	1
Director, Aeronautical & Maritime Research Laboratory	1
Electronics and Surveillance Research Laboratory	
Director Electronics Research Laboratory	1
Chief Information Technology Division	1
Research Leader Command & Control and Intelligence Systems	1
Research Leader Military Computing Systems	1
Research Leader Command, Control and Communications	1
Executive Officer, Information Technology Division	Doc Control sheet
Head, Information Architectures Group	1
Head, C3I Systems Engineering Group	1
Head, Information Warfare Studies Group	Doc Control sheet
Head, Software Engineering Group	2
Head, Trusted Computer Systems Group	Doc Control sheet
Head, Advanced Computer Capabilities Group	Doc Control sheet

Head, Systems Simulation and Assessment Group	Doc Control sheet
Head, Intelligence Systems Group	Doc Control sheet
Head Command Support Systems Group	1
Head, C3I Operational Analysis Group	Doc Control sheet
Head Information Management and Fusion Group	Doc Control sheet
Head Human Systems Integration Group	Doc Control sheet
Publications and Publicity Officer, ITD	1
Gina Kingston	2
Martin Burke	2
Peter Fisher	1

DSTO Library

Library Fishermens Bend	1
Library Maribyrnong	1
Library DSTOS	2
Library, MOD, Pyrmont	Doc Control sheet

Forces Executive

Director General Force Development (Sea),	Doc Control sheet
Director General Force Development (Land),	Doc Control sheet
Director General Force Development (Air),	Doc Control sheet

Army

ABCA Office, G-1-34, Russell Offices, Canberra	4

S&I Program

Defence Intelligence Organisation	1
Library, Defence Signals Directorate	Doc Control sheet

B&M Program (libraries)

OIC TRS, Defence Central Library	1
Officer in Charge, Document Exchange Centre (DEC),	1
US Defence Technical Information Center,	2
UK Defence Research Information Centre,	2
Canada Defence Scientific Information Service,	1
NZ Defence Information Centre,	1
National Library of Australia,	1

Universities and Colleges

Australian Defence Force Academy	1
Library	1
Dr Chris Lokan, Australian Defence Force Academy	1
Head of Aerospace and Mechanical Engineering	1
Senior Librarian, Hargrave Library, Monash University	1
Librarian, Flinders University	1
Professor Ross Jeffery, University of New South Wales	1
Professor Ray Offen, JRCASE, Macquarie University	1
Professor Richard Jarret, University of Adelaide	1
Simon Timcke, University of Adelaide	1

Other Organisations	
NASA (Canberra)	1
AGPS	1
State Library of South Australia	1
Parliamentary Library, South Australia	1

OUTSIDE AUSTRALIA

Abstracting and Information Organisations	
INSPEC: Acquisitions Section Institution of Electrical Engineers	1
Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
American Society for Metals	1
Documents Librarian, The Center for Research Libraries, US	1
Information Exchange Agreement Partners	
Acquisitions Unit, Science Reference and Information Service, UK	1
Library - Exchange Desk, National Institute of Standards and Technology, US	1
SPARES	10
Total number of copies:	**72**

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

A Review of Software Sizing for Cost Estimation

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Gina Kingston and Martin Burke

5. CORPORATE AUTHOR

Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury SA 5108

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TR-0360	AR-009-738	Technical Report	June 1996

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
N9505/10/104	93/349	DST	102	86

13. DOWNGRADING/DELIMITING INSTRUCTIONS
N/A

14. RELEASE AUTHORITY

Chief, Information Technology Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE CENTRE, DIS NETWORK OFFICE, DEPT OF DEFENCE, CAMPBELL PARK OFFICES, CANBERRA ACT 2600

16. DELIBERATE ANNOUNCEMENT

No limitations

17. CASUAL ANNOUNCEMENT Yes

18. DEFTEST DESCRIPTORS

iMAPS
Software costs
Estimation
Computer Software
Software Engineering

19. ABSTRACT

A variety of Software Sizing approaches have been conjectured in the literature. While many of these approaches have been used for Software Cost Estimation, none were specifically designed for Software Costing. This paper reviews current methods of Software Sizing to determine their suitability for Software Cost Estimation. The approach differs from previous approaches in that it is based on a framework of evaluation criteria including Measurement Theoretic as well as practical considerations.